
UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# Uma abordagem unificada para especificar e checar restrições em múltiplas linguagens de programação por meio de um analisador estático no contexto de um juiz *on-line*

Dissertação de Mestrado

Kleber Tarcísio Oliveira Santos


São Cristóvão – Sergipe

2018

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Kleber Tarcísio Oliveira Santos

# Uma abordagem unificada para especificar e checar restrições em múltiplas linguagens de programação por meio de um analisador estático no contexto de um juiz *on-line*

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de mestre em Ciência da Computação.

Orientador(a): Prof. Dr. Alberto Costa Neto

São Cristóvão – Sergipe

2018

Kleber Tarcísio Oliveira Santos

Uma abordagem unificada para especificar e checar restrições em múltiplas linguagens de programação por meio de um analisador estático no contexto de um juiz *on-line*/ Kleber Tarcísio Oliveira Santos. – São Cristóvão – Sergipe, 2018-

115 p. : il.

Orientador: Prof. Dr. Alberto Costa Neto

Dissertação de Mestrado – UNIVERSIDADE FEDERAL DE SERGIPE
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO, 2018.

1. Juiz *on-line*. 2. Ensino de programação. 3. Avaliação automática de programação. 4. Análise estática de código-fonte.

CDU 004.41

**Kleber Tarcísio Oliveira Santos**

# Uma abordagem unificada para especificar e checar restrições em múltiplas linguagens de programação por meio de um analisador estático no contexto de um juiz *on-line*

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Sergipe como requisito parcial para a obtenção do título de mestre em Ciência da Computação.

Trabalho aprovado. São Cristóvão – Sergipe, 28 de Fevereiro de 2018:

---

**Prof. Dr. Alberto Costa Neto**
Orientador

---

**Prof. Dr. Rodrigo de Barros Paes, Membro**
Universidade Federal de Alagoas (UFAL)

---

**Prof. Dr. André Britto de Carvalho, Membro**
Universidade Federal de Sergipe (UFS)

São Cristóvão – Sergipe
2018

*I dedicate this work to God, all my parents, professors and friends who gave me the opportunity to learn.*

# Agradecimentos

Agradeço primeiramente a Deus, por ser a verdadeira essência da minha vida, por ter dado a toda humanidade o maior gesto de amor por meio de seu Filho unigênito e por me permitir conhecer pessoas de nobreza incalculável.

Aos meus pais, por terem me ensinado a verdade e os valores que norteiam meu caráter, por serem presentes e por perceberem o valor da educação na vida dos filhos.

Aos meus irmãos, pela confiança, motivação e entusiasmo. Através dos meus irmãos pratiquei os primeiros ensinamentos recebidos pelos meus pais.

Ao professor Dr. Alberto Costa Neto, pela paciência e principalmente pela enorme partilha de conhecimento. Agradeço por todas as correções, por todas as discussões de pesquisa e por me mostrar que a Ciência da Computação é muito maior do que qualquer grade curricular. Sou muito grato por tudo. Muito obrigado!

A todos os meus professores pelos ensinamentos e críticas que me fizeram progredir.

Aos meus amigos pelo companheirismo.

*"Feliz é a pessoa que encontrou a sabedoria e alcançou o entendimento! Porque a sabedoria é mais valiosa do que a prata, e rende mais do que o ouro. Ela é mais preciosa do que as pérolas; nenhum objeto desejado a ela se compara"Pr 3,13-15*

# Resumo

O processo de ensino e aprendizagem da programação de computadores é uma tarefa complexa que requer bastante prática e criatividade. Geralmente há inúmeras soluções para um mesmo problema. Por isso, o aluno precisa que suas soluções sejam avaliadas rapidamente visando um aprendizado mais ágil e eficaz. Para enfrentar esses desafios, os professores e alunos podem contar com recursos provenientes da evolução da Tecnologia da Informação e Comunicação. Os ambientes de aprendizagem virtual e os sistemas de juiz *on-line* são alternativas atrativas utilizadas nesse contexto. Este trabalho apresenta uma abordagem unificada de especificação e checagem de restrições de código-fonte apoiada por um analisador estático. Apesar das ferramentas atuais serem capazes de indicar se o programa produziu a saída esperada a partir de uma entrada fornecida, nem todas são capazes de determinar se o aluno utilizou (ou não) determinada construção de linguagem de programação, como por exemplo criar uma função e utilizá-la no programa. Entre as que são capazes, existem problemas que foram sanados na abordagem proposta neste trabalho, como: facilidade de uso, abordagem unificada e grau de flexibilidade. Além disto, este trabalho conta com uma análise da base de dados do The Huxley com o objetivo de descobrir quais são as principais restrições de código-fonte utilizadas pelos professores e atendidas pelos alunos. Esta análise foi feita com os dados obtidos da aplicação do analisador estático de código-fonte desenvolvido e em conjunto com um *survey* aplicado aos professores de introdução à programação com o propósito de conhecer as principais restrições que seriam utilizadas por eles se possuíssem uma ferramenta de especificação e checagem de restrições.

**Palavras-chave: juiz *on-line*, ensino de programação, avaliação automática de programação, análise estática de código-fonte**

# Abstract

The teaching and learning process of computer programming is a complex task which requires a lot of practice and creativity. Usually, there are numerous solutions to the same problem. Therefore, the student needs that his solutions are evaluated quickly for a faster and effective learning. To face these challenges, teachers and students can rely on resources from the evolution of Information and Communication Technology. Virtual learning environments and online judge systems are attractive alternatives used in this context. This work presents a unified approach to specify and check source code restrictions supported by a static analyzer. Although current tools are able to indicate if the program produced the expected output from a given input, not all are able to determine if the student used (or not) a given programming language construct, such as creating a function and using it in the program. Among those that are capable, there are problems that were solved in the approach proposed in this work, such as: ease of use, unified approach and degree of flexibility. In addition, this work presents an analysis of the database of The Huxley with the purpose of discovering the main restrictions of source code used by the teachers and attended by the students. This analysis was done based on data obtained from the use of the developed static analyzer and in conjunction with a survey applied to the teachers of introduction to programming with the purpose of knowing the main restrictions that would be used by them if they had a tool to specify and check restrictions.

**Keywords: judge on-line, teaching programming, automated assessment of programming, static analysis of source code**

# Lista de ilustrações

# Lista de tabelas

# Lista de Quadros

# Lista de códigos

# Lista de abreviaturas e siglas

| | |
|---|---|
| API | Application Programming Interface |
| AVA | Ambiente Virtual de Aprendizagem |
| AST | Abstract Syntax Tree |
| IDE | Integrated Development Environment |
| MOODLE | Modular Object Distance Learning |
| ND | Nível de Dificuldade |
| UFAL | Universidade Federal de Alagoas |
| UFS | Universidade Federal de Sergipe |
| ETL | Extract, Transform, Load |
| SBC | Sociedade Brasileira de Computação |

# Sumário

# 1 Introdução

Nos últimos anos, os conhecimentos de programação de computador vêm adquirindo maior importância nas grades curriculares nas áreas das ciências exatas, tornando-se conhecimento obrigatório e necessário (MARCOLINO; BARBOSA, 2015). No entanto, o ensino de programação é uma tarefa complexa (OLIVEIRA; MOTA; OLIVEIRA, 2015), pois diversos problemas são enfrentados pelos alunos e professores, além de ser uma atividade que requer criatividade e capacidade de trabalhar com recursos limitados de programação.

Para auxiliar no processo de ensino e aprendizagem de diversas áreas, e não apenas no âmbito dos cursos iniciais de programação, surgiram os Ambientes Virtuais de Aprendizagem. Conforme definido em (CARDOSO, 2010), Ambiente Virtual de Aprendizagem, ou AVA, consiste num sistema computacional, destinado à construção de estratégias pedagógicas e montagem de cursos *on-line* ou para apoio em cursos presenciais, que possibilita o armazenamento, o gerenciamento, a atualização/edição e o fluxo de recursos e conteúdos educacionais. Em um AVA, presume-se e permite-se a autonomia intelectual do aluno, o que o torna um instrumento muito interessante nos novos modelos pedagógicos que envolvem a participação ativa dos aprendizes, sem excluir a presença dos professores.

O Ambiente Colaborativo de Aprendizagem (e-Proinfo) (E-PROINFO, 2016), exemplo de AVA, é um ambiente virtual colaborativo de aprendizagem que permite a concepção, administração e desenvolvimento de diversos tipos de ações, como cursos a distância, complemento a cursos presenciais, projetos de pesquisa, projetos colaborativos e diversas outras formas de apoio a distância e ao processo ensino-aprendizagem.

O Moodle (MOODLE, 2016), por exemplo, é uma plataforma de aprendizagem concebida para proporcionar aos educadores, administradores e alunos um único sistema robusto, seguro e integrado para criar ambientes de aprendizagem personalizados. O Moodle não apenas provê um ambiente de aprendizagem, mas também promove uma comunidade com os seus usuários.

Além dos AVAs, existem ferramentas específicas para as disciplinas de programação que

são conhecidas como Juízes *On-line*. Um juiz *on-line* é um sistema composto por um repositório de problemas de programação disponíveis para os usuários submeterem suas respostas em forma de código-fonte. Alguns desses sistemas ainda possuem um fórum de discussões de estratégias para solucionar determinado problema. Além disso, os alunos possuem a liberdade para escolher em qual linguagem de programação submeterá a sua resposta.

Nesses sistemas, os professores podem cadastrar ou selecionar um conjunto de problemas para que sejam resolvidos pelos alunos. Quando um aluno submete uma resposta, o sistema executa o código-fonte diversas vezes de acordo com a quantidade de casos de teste que o problema possui. A resposta é aceita quando fornece os resultados esperados para cada respectivo caso de teste. Entretanto, quando ela não é aceita o sistema retorna algum destes erros: *Incorrect output, time-limit exceeded, runtime error, compilation error, output format error e contact staff*. Há diversos sistemas de juízes *on-line*, como: URI Online Judge (URI, 2016), UVA Online Judge(UVA, 2016), SPOJ Online Judge(JUDGE, 2016), The Huxley (PAES et al., 2013), Feeper (FEEPER, 2016). Este tópico será explorado na seção 2.1.

Os principais benefícios ao se adotar um sistema de correção automática *on-line* são:

- Menor esforço, uma vez que conta com o auxílio da ferramenta;
- Melhor administração dos estudantes e de suas tarefas;
- Melhor rastreamento individual dos estudantes;
- Melhor qualidade de ensino, devido ao maior tempo de prática;
- Mais tempo para contato com os estudantes.

Apesar da existência de diversos sistemas de juiz *on-line* que facilitam o processo de ensino e aprendizagem nas disciplinas de programação, ainda há o que aprimorar nesse seguimento devido às dificuldades persistentes. Este trabalho concentra-se na dificuldade de forçar o aluno a resolver os problemas conforme o que foi especificado. É importante ressaltar que neste momento o objetivo não é identificar se a solução é eficiente em termos de complexidade de algoritmo, como é exigido em maratonas de programação ou até mesmo em relação à quantidade de memória utilizada pelo programa. Na verdade, como o público alvo são alunos que estão aprendendo a programar e por isso estão ainda aprendendo a utilizar construções simples de programação como estrutura condicional, estrutura de repetição, função, função recursiva, declaração de variáveis e outros, o objetivo é forçar o aluno a utilizar construções especificadas no problema, as quais chamamos de restrições estruturais, para resolvê-lo.

Para exemplificar, o quadro 1.1 mostra um problema (HUXLEY., 2014) cadastrado no The Huxley que exige que a solução possua uma função que receba dois parâmetros e possua retorno. Obviamente, existem diversas soluções para esse problema que não necessariamente precisam de função com dois parâmetros e com retorno. Entretanto, o objetivo é fazer o aluno

| **Descrição** | |
|---|---|
| A Escola de Música Melodia é bastante conceituada na cidade e forma grandes profissionais. Para isso, exige que seus alunos tenham no máximo 10 faltas por semestre, e que obtenham média mínima 7 para serem aprovados. Aqueles que não excedem o limite de faltas e conseguem média igual ou superior a 9,5 são aprovados com louvor.<br>**Escreva uma função chamada ClassificaAluno** que receba como entrada a media e a quantidade de faltas de um aluno e retorne sua situação ao final do semestre. | |
| **Formato de entrada** | |
| Um número real e um número inteiro, nessa ordem | |
| **Formato de saída** | |
| Uma *String* em letras maiúsculas (APROVADO COM LOUVOR ou APROVADO ou REPROVADO ou REPROVADO POR FALTAS | |
| **Entrada** | **Saída** |
| 6.9<br>10 | REPROVADO |

Quadro 1.1 – Problema Escola de Música

exercitar a criação de funções e saber como utilizá-las. A listagem 1.1 apresenta uma solução proposta por um aluno que foi aceita pelo The Huxley, todavia não possui as restrições estruturais que a descrição do problema exige. Casos como esse acontecem porque não há uma verificação das construções que existem no código do aluno. Apenas é feita uma comparação do resultado que foi obtido pela solução submetida com as respostas cadastradas pelo professor para cada caso de teste.

A listagem 1.2 mostra uma solução proposta por outro aluno que, além de apresentar o resultado correto, possui uma função que recebe dois parâmetros e possui retorno, ou seja, obedece as restrições do problema.

```
1  n = float(raw_input())
2  f = int(raw_input())
3  if f <= 10 and n >= 9.5:
4      print 'APROVADO COM LOUVOR'
5  if f <= 10 and 7 <= n<9.5:
6      print 'APROVADO'
7  if f > 10:
8      print 'REPROVADO POR FALTAS'
9  if f <= 10 and n < 7:
10     print 'REPROVADO'
```

Listagem 1.1 – Solução incorreta do problema Escola de Música

```
1  media  = float(raw_input())
2  faltas = int(raw_input())
3  def ClassificaAluno(media,faltas):
4    if faltas <= 10:
5      if media >= 9.5:
6        return 'APROVADO COM LOUVOR'
```

```
7       elif media >= 7:
8         return 'APROVADO'
9       else:
10        return 'REPROVADO'
11    if faltas > 10:
12        return 'REPROVADO POR FALTAS'
13  print ClassificaAluno(media,faltas)
```

Listagem 1.2 – Solução correta do problema Escola de Música

O quadro 1.2 apresenta outro problema (MORCOURT., 2016) do The Huxley. Nesse problema é solicitado ao aluno que programe a operação de exponenciação com auxílio de alguma estrutura de repetição. Nesse caso o objetivo do professor não é apenas fazer o aluno utilizar uma determinada estrutura de programação, mas também desenvolver a lógica de programação implícita que o problema possui.

| **Descrição** | |
|---|---|
| Faça um programa em Java que peça dois números, base e expoente, calcule e mostre o primeiro número elevado ao segundo número. **Utilize estruturas de repetição** para resolver o problema. | |
| **Formato de entrada** | |
| Base e expoente. | |
| **Formato de saída** | |
| Resultado do cálculo. | |
| **Entrada** | **Saída** |
| 3<br>4 | Informe a base:<br>Informe o expoente:<br>Resultado: 81 |
| **Entrada** | **Saída** |
| 2<br>0 | Informe a base:<br>Informe o expoente:<br>Resultado: 1 |

Quadro 1.2 – Problema da Base e expoente

A listagem 1.3 apresenta uma solução que foi submetida e aceita pelo The Huxley. Essa solução não utiliza uma estrutura de repetição, apenas uma função *pow()* que realiza a exponenciação. Nesse caso, além de o aluno não ter exercitado a utilização da estrutura, é possível que ele não tenha desenvolvido a lógica de programação que o problema exige.

```
1 print 'Informe a base:'
2 base = int(raw_input())
3 print 'Informe o expoente:'
4 expoente = int(raw_input())
5 resultado = pow(base,expoente)
6 print 'Resultado:', resultado
```

Listagem 1.3 – Solução incorreta do problema Base e expoente

A listagem 1.4 mostra uma solução correta que utiliza uma estrutura de repetição no cálculo da exponenciação.

```
1 print 'Informe a base:'
2 base = int(raw_input())
3 print 'Informe o expoente:'
4 expoente = int(raw_input())
5 resultado = 1
6 for i in range(expoente):
7         resultado = resultado * base
8 print 'Resultado:', resultado
```

Listagem 1.4 – Solução correta do problema Base e expoente

Apesar de os Juízes *On-line* ajudarem bastante no processo de aprendizagem das disciplinas iniciais de programação, casos como esses só poderiam ser percebidos se o professor verificasse todas as soluções apresentadas por todos os alunos da turma. Entretanto, em alguns casos essa tarefa pode demandar grande esforço manual do professor, quando na verdade o ideal é que fosse feito pela ferramenta.

## 1.1 Objetivos

O objetivo geral deste trabalho consiste em realizar uma melhoria no processo de correção de problemas utilizado pelos juízes *on-line* no contexto das disciplinas iniciais de programação. Além disso, a solução deve ser fácil de ser usada pelo professor, ou seja, não deve requerer que o mesmo utilize recursos complexos para declarar as restrições para a solução do problema.

### 1.1.1 Objetivos Específicos

Foram estabelecidos os seguintes objetivos específicos:

- Realizar um levantamento das reais necessidades de especificação de restrições;
- Criar uma abordagem para especificação de requisitos (restrições) de construções de programação;
- Desenvolver um Analisador Estático de código-fonte;
- Verificar se as submissões propostas pelos alunos seguem as restrições impostas pelos professores;
- Criar uma *Application Programming Interface* (API) para comunicação com o Analisador Estático;
- Customizar o The Huxley para suportar o novo processo de correção de problemas.
- Aplicar um *Survey* para tentar validar a proposta desenvolvida.

## 1.2 Metodologia

Esta seção apresenta um resumo dos métodos e critérios utilizados neste trabalho para alcançar seus objetivos. Detalhes completos são encontrados no Capítulo 3.

Inicialmente foi realizado um levantamento das reais necessidades dos professores de maneira manual por meio da leitura do enunciado de diversos problemas do The Huxley. Após verificar as necessidades preliminares, foi desenvolvido um Analisador Estático de código-fonte com auxílio do framework ANTLR 4 (PARR, 2013) para as linguagens Java e Python. Estas duas linguagens foram escolhidas por possuírem perfis diferentes. Java é estaticamente tipada, enquanto Python é dinamicamente tipada.

Após o desenvolvimento do Analisador Estático, foi implementado um serviço web em Python para obter as submissões dos alunos de problemas que exigiam algum tipo de restrição de código-fonte. Esses dados estão na forma de arquivos texto disponíveis no Dropbox, em Java [1] e em Python [2]. O *survey* foi desenvolvido pela plataforma Google Forms e aplicado para 38 professores que possuem experiência em turmas introdutórias de programação.

## 1.3 Organização da dissertação

Este trabalho está organizado em 7 capítulos e 3 apêndices. Neste primeiro capítulo, foram apresentados o problema e o objetivo da pesquisa. Os demais capítulos estão divididos da seguinte forma:

- O capítulo 2 apresenta a fundamentação teórica para esta pesquisa. Ela contém o conceito e exemplos de juízes on-line, os dois principais tipos de abordagens que existem para resolver o problema em análise;
- No capítulo 3 é apresentada a metodologia adotada para realização deste trabalho.
- No capítulo 4 é apresentada uma análise da base de dados do The Huxley com o objetivo de verificar quais são as principais restrições utilizadas pelos professores e se os alunos seguem essas restrições.
- No capítulo 5 é apresentada com detalhes a abordagem proposta por este trabalho e as decisões de projeto tomadas no decorrer da pesquisa.
- O capítulo 6 possui os trabalhos relacionados e aplicação de um *survey*. Estes dois componentes foram utilizados para avaliar a proposta desenvolvida.
- O capítulo 7 possui a conclusão deste trabalho, apontando quais objetivos foram alcançados e apresentando os prováveis trabalhos futuros.

[1] https://www.dropbox.com/sh/c3ynb5f3va9vo3i/AABMw4E1c3ry91nRtq1cPhP2a?dl=0
[2] https://www.dropbox.com/sh/htc2d8ndunvne74/AACCmWjFVJu3vNhh3lKX1JWha?dl=0

- O apêndice A possui uma lista de problemas difíceis que não foram considerados para a análise realizada no capítulo 4.
- O apêndice B possui o manual com instruções para utilização da abordagem desenvolvida nesta pesquisa.
- O apêndice C apresenta em detalhes os passos que devem ser seguidos para evoluir o Analisador Estático desenvolvido.

# 2

# Juízes on-line e Avaliação Automática

Neste capítulo é apresentado um levantamento bibliográfico com os principais trabalhos e conceitos teóricos necessários para fundamentar esta pesquisa. Na primeira seção são apresentados os conceitos e exemplos de juízes *on-line*, na segunda seção são apresentadas as abordagens utilizadas para criação de sistemas de avaliação automática.

## 2.1 Juízes *on-line*

Juízes *on-line* são sistemas web que fornecem um repositório de diversos problemas de programação e estão disponíveis para que estudantes possam submeter as possíveis soluções destes em forma de código-fonte, numa linguagem de programação. Essas soluções submetidas são avaliadas e as respostas são retornadas com a situação da correção (correta ou incorreta) para o dado problema. No caso da solução submetida estar incorreta, ele retorna as informações sobre o erro (OLIVEIRA; MOTA; OLIVEIRA, 2015; BEZ; TONIN; RODEGHERI, 2014; SUN; Bofang Li, 2014; PAES et al., 2013; GIMÉNEZ; PETIT; ROURA, 2012).

Utilizando juízes *on-line*, os professores podem acompanhar individualmente cada um dos alunos, e estes podem realizar as atividades práticas de programação de qualquer lugar e a qualquer momento, obtendo retorno em tempo real sobre as correções de suas soluções implementadas (BEZ; TONIN; RODEGHERI, 2014; SUN; Bofang Li, 2014; GIMÉNEZ; PETIT; ROURA, 2012; PAES et al., 2013; WU; Shuangping Chen, 2012). Isso faz aumentar o interesse do aluno e consequentemente melhoram sua capacidade de resolver problemas e de programar (PAES et al., 2013; WU; Shuangping Chen, 2012).

### 2.1.1 Feeper

O Feeper é um ambiente virtual para apoiar o processo de ensino-aprendizagem de programação. O nome é um acrônimo entre as palavras "***fee****dback*" e "**per**sonalizado". Ele foi

projetado para melhorar a comunicação entre aluno e professor. Nesse ambiente, quando o estudante envia alguma mensagem para o professor, essa mensagem é vinculada a um trecho de código-fonte, o que permite destacar a linha a qual está ocorrendo sua dúvida. Além disso, para responder aos exercícios, o aluno utiliza um editor de código-fonte, podendo criar múltiplas classes e também, caso necessário, realizar o *upload* de classes existentes.(ALVES; JAQUES, 2014)

Nesta ferramenta a validação das soluções do aluno é feita de forma imediata, utilizando a tecnologia de Juiz *on-line* para executar testes automáticos a fim de determinar se a solução do aluno está correta ou não.

### 2.1.2 Jutge.org

Jutge.org é um juiz *on-line* de programação educacional gratuito em que os alunos podem ter acesso a cerca de 1.500 problemas classificados que podem ser resolvidos através de 20 diferentes linguagens de programação (C++, C, Java, Python, Scheme, Pascal, etc). O repositório de problemas do Jutge.org é organizado por temas e oferecido na língua inglesa, mas existem as declarações dos problemas também disponibilizados em catalão (GIMÉNEZ; PETIT; ROURA, 2012).

O Jutge.org é integrado ao Moodle e fornece um AVA para o ensino de programação. Neste ambiente os alunos podem se inscrever nos cursos e os professores podem supervisioná-los (GIMÉNEZ; PETIT; ROURA, 2012).

### 2.1.3 The Huxley

The Huxley é uma ferramenta web que permite aos alunos submeterem código-fonte em diversas linguagens de programação como respostas a exercícios de uma base de centenas de problemas. Para cada submissão, o aluno recebe *feedback* da correção automática pelo sistema através de análise sintática do código e dos testes de aceitação. Foi pensado para auxiliar o aluno e o professor dentro e fora de sala de aula (PAES et al., 2013).

Com o The Huxley, o professor acompanha o desempenho de seus alunos: quantidade de problemas resolvidos, porcentagem de acertos/erros, tipos de problemas com mais erros, detecção de plágio e erros específicos de cada aluno. Professores e alunos possuem diferentes visões, ambas acessíveis através de *dashboard*, que permite uma visualização global do status de exercícios, avaliações e conteúdo (PAES et al., 2013).

Além dessas características, possui também:

- Acompanhamento das atividades do aluno pelo professor;
- Problemas separados por categorias e por níveis de dificuldades;

- *Ranking* por problema e por linguagem de programação;
- Visualização e edição do código-fonte diretamente no navegador;
- Visualização das linhas do código-fonte com erro quando recebe como resposta o Erro de Compilação;
- Visualização de detalhes sobre erros encontrados durante a execução da solução;
- Indica o percentual de casos de teste que falharam, quando da submissão de uma solução incorreta para julgamento;
- Avaliações automatizadas, ou seja, é possível definir data e hora de início de disponibilização da prova, bem como seu tempo de expiração;
- Histórico de submissão de uma solução realizada pelo aluno.
- Recurso de enviar uma mensagem ao professor vinculada a um trecho específico de código-fonte. Esse recurso permite que o professor entenda com mais facilidade qual a dúvida do aluno.

Existe uma série de juízes *on-line* além dos 3 apresentados acima, como URI [1], SPOJ [2], UVA [3] e Codeforces [4]. Essas plataformas são usadas principalmente por alunos que desejam treinar para competições de Maratona de Programação. Esses juízes *on-line* possuem características semelhantes aos já discutos neste capítulo, por exemplo: possuem um repositório de problemas e aceitam respostas em diversas linguagens de programação.

## 2.2 Abordagens

Ao longo dos anos foram desenvolvidas várias técnicas para implementar um sistema de correção automática ou semi-automática de problemas de programação. De maneira geral, essas técnicas são baseadas em duas abordagens: análises dinâmica e estática. As duas abordagens se complementam e geralmente são utilizadas em conjunto em um mesmo sistema a fim de deixá-lo mais robusto.

### 2.2.1 Análise Dinâmica

Esta abordagem envolve a execução do código-fonte do estudante e geralmente é utilizada para verificar se o resultado da solução submetida está de acordo com o que o professor esperava.

[1] https://urionlinejudge.com.br/
[2] http://br.spoj.com/
[3] https://uva.onlinejudge.org
[4] http://codeforces.com/

Como esta abordagen necessita da execução da solução, a sobrecarga de processamento pode acontecer com o servidor do sistema. Algumas medidas podem ser tomadas para evitar que os programas enviados pelos estudantes não contenham laços infinitos ou erros críticos que causem a indisponibilidade do servidor (RAHMAN; NORDIN, 2007).

### 2.2.2 Análise Estática

Esta abordagem é utilizada para coletar informações sobre o código-fonte sem precisar executá-lo (ALA-MUTKA, 2005). No âmbito do desenvolvimento de sistemas, a análise estática (teste caixa-branca) é utilizada em conjunto com testes caixa-preta com o objetivo de aumentar a qualidade do software. Algumas ferramentas que realizam análise estática são: FindBugs [1], Checkstyle [2] e PMD [3].

As próximas seções apresentam algumas técnicas de análise estática. Um estudo mais aprofundado dessas e de outras técnicas pode ser encontrado em (RAHMAN; NORDIN, 2007).

#### 2.2.2.1 Análise de Métricas de Software

A medição de software preocupa-se com a derivação de um valor numérico ou o perfil para um atributo de um componente de software, sistema ou processo. Comparando esses valores entre si e com os padrões que se aplicam a toda a organização, você pode ser capaz de tirar conclusões sobre a qualidade do software ou avaliar a eficácia dos métodos, das ferramentas e dos processos de software. (SOMMERVILLE, 2011)

Algumas das métricas estáticas de um software são: *Fan-in/Fan-out*, comprimento de código, complexidade ciclomática, comprimento de identificadores, profundiade de aninhamento condicional, índice *Fog* e outras. Em sistemas orientados a objetos pode-se verificar os níveis de acoplamento e coesão entre classes. No desenvolvimento de sistemas é recomendado que o mesmo possua um baixo nível de dependência (acoplamento) e um alto nível de divisão de responsabilidades (coesão) entre as classes.

#### 2.2.2.2 Detecção de erros

Por meio da análise estática também é possível verificar se há erros sintáticos na solução do aluno e dar alguma dica de como corrigi-lo. Um dos erros mais comuns para programadores iniciantes em C e Java é começar a escrever um bloco de código e esquecer o token } para fechá-lo. Essa situação geralmente é detectada por modernas IDE (Integrated Development Enviroment), que auxiliam o programador oferecendo sugestões de correção.

[1] http://findbugs.sourceforge.net/
[2] http://checkstyle.sourceforge.net/
[3] https://pmd.github.io/

#### 2.2.2.3 Análise de similaridade

A análise estática também permite verificar o grau de similaridade entre dois ou mais algoritmos. No contexto deste trabalho, estas técnicas podem ser utilizadas para verificar se um aluno copiou ou possui uma solução muito próxima de outro aluno. Outra aplicabilidade é verificar se a solução do aluno é próxima da solução exigida pelo professor, nesse caso o professor precisa fornecer como entrada um conjunto de soluções pré-definidas.

Para medir o nível de similaridade entre as soluções, algumas técnicas transformam o código-fonte em um representação mais simples chamada de pseudo-código abstrato, essa nova representação possui apenas as partes mais importantes e básicas do algoritmo. Após essa transformação as ASTs de cada solução são comparadas e definido um grau de similaridade. Se esta técnica for utilizada para verificar se a solução do aluno é próxima da esperada pelo professor, é importante verificar o nível de complexidade do problema, pois quanto mais complexo é um problema mais respostas diferentes ele pode ter. Nesse caso, o professor precisa imaginar as diversas soluções possíveis para o exercício. Diante disso, esta técnica possui resultados mais eficazes com problemas simples que exigem poucas linhas de código.

#### 2.2.2.4 Análise de palavra-chave

Esta técnica permite verificar se a solução do aluno possui um conjunto definido de palavras-chave. Scheme-robo (SAIKKONEN; MALMI; KORHONEN, 2001), Portugol Studio (STUDIO, 2017) e a abordagem desenvolvida neste trabalho permitem este tipo de análise. Ela pode ser utilizada tanto para forçar que o aluno resolva um problema com um conjunto específico de funcionalidade quanto para proibir o mesmo de utilizar esse mesmo conjunto de funcionalidade.

De acordo com o problema de pesquisa apresentado no capítulo 1 e o mapeamento de restrições apresentado no capítulo 4, é possível perceber que este trabalho está concentrado em uma abordagem estática de avaliação de código-fonte. Este trabalho não se concentra na execução da solução, pois foca nas relações entre as construções da linguagem de programação.

# 3
# Metodologia

Este capítulo apresenta os métodos e critérios utilizados para: obtenção dos dados da plataforma The Huxley, desenvolvimento do *survey* e escolha das linguagens de programação. Além disso, apresenta como se deu o desenvolvimento da pesquisa e quais foram as principais ferramentas utilizadas.

## 3.1 Obtenção dos dados do The Huxley

Os dados (submissões) da base do The Huxley passaram por um processo de ETL *(Extract, Transform, Load)* para que pudessem ser utilizados. A etapa de extração ocorreu via *web service*, o cliente foi implementado em Python e utilizou serviços disponibilizados pela plataforma The Huxley.

A etapa de transformação foi responsável por deixar as submissões de modo que pudessem ser avaliadas pelo analisador estático desenvolvido. Dessa forma, foi necessário alterar apenas alguns comandos simples de algumas submissões feitas em Python 2.7 para que o analisador estático (desenvolvido para Python 3) pudesse entendê-las. Esse processo não foi necessário para a linguagem Java porque a única versão utilizada pelo The Huxley é a mesma utilizada nesta abordagem (Java 7).

Este trabalho não utilizou um SGBD (Sistema Gerenciador de Banco de Dados) para realizar o carregamento das submissões, entretanto as mesmas estão disponíveis no Dropbox em arquivos TXT, em Java[1] e Python[2].

[1] https://www.dropbox.com/sh/c3ynb5f3va9vo3i/AABMw4E1c3ry91nRtq1cPhP2a?dl=0
[2] https://www.dropbox.com/sh/htc2d8ndunvne74/AACCmWjFVJu3vNhh3lKX1JWha?dl=0

## 3.2 Escolha das linguagens de programação

Esta abordagem implementou funcionalidades em duas linguagens de programação: Python e Java. Alguns critérios foram levados em consideração na escolha das linguagens.

O primeiro critério foi escolher linguagens que fossem diferentes quanto a tipagem. Se por um lado Java é fortemente tipada, por outro Python é dinamicamente tipada. Essa diferença nos permite mostrar que nossa abordagem funciona simultaneamente para duas linguagens com perfis diferentes.

O segundo critério foi escolher linguagens estruturais ou orientadas a objetos, pois as linguagens que utilizam estes paradigmas são utilizadas com mais frequência em turmas introdutórias de programação do que alguma linguagem com paradigma funcional.

Além disso, de acordo com alguns trabalhos (DIERBACH, 2014) e (SHEIN, 2015), a linguagem de programação Python vem ganhando popularidade nos últimos anos nas disciplinas de introdução à programação por ser mais simples de usar e também por aumentar a satisfação dos estudantes e professores.

## 3.3 Survey

Um *survey* com 29 perguntas foi desenvolvido por meio do Google Forms[3] e disponibilizado via *e-mail* para professores cadastrados na lista da SBC (Sociedade Brasileira de Computação). Apenas professores que possuem experiência no ensino de introdução à programação puderam responder.

O objetivo do *survey* foi descobrir quais as principais restrições que os professores usariam se possuíssem uma ferramenta para especificação e checagem de restrições de código-fonte. Após a obtenção dos dados foi feita uma comparação com as funcionalidades desenvolvidas nesta abordagem para verificar se as principais necessidades dos professores seriam atendidas.

## 3.4 Ferramentas utilizadas

As linguagens de programação utilizadas neste trabalho foram Python e Java. A linguagem Java foi utilizada para desenvolvimento do analisador estático e a linguagem Python para criação do consumidor do *web service*.

As principais ferramentas de desenvolvimento utilizadas foram: Eclipse [4], Junit [5], Git [6],

[3] https://www.google.com/forms/about/
[4] http://www.eclipse.org/home/index.php
[5] http://junit.org/
[6] https://git-scm.com/

Bitbucket [7] e Maven [8].

O Eclipse é uma IDE *open source* para desenvolvimento Java, porém suporta outras linguagens a partir de plug-ins como C/C++, PHP, Python, Scala e outras. Além disso, possui nativamente recursos de análise estática, teste unitário (JUnit), depuração, refatoração, *build*, compilação e execução de projetos Java.

O sistema de controle de versão distribuído Git também é *open source* e demasiadamente utilizado no desenvolvimento de software convencional. Diferentemente dos sistemas centralizados como CVS e SVN [9], em que o histórico de todas as mudanças está concentrado em um repositório central, no Git o usuário possui uma cópia completa de todo o histórico, tags e branches do projeto. Por causa disso, nos sistemas de versionamento distribuído, *commit* e *checkout* são feitos no repositório local de cada ambiente. Após o trabalho realizado localmente, os desenvolvedores devem utilizar os comandos *push* (envia atualizações) e *pull* (recebe atualizações).

O Bitbucket é um serviço de hospedagem de projetos controlados por meio dos sistemas de controle de versão distribuídos Git ou Mercurial [10]. Atualmente o serviço é utilizado por mais de 6 milhões de desenvolvedores divididos em mais de 1 milhão de equipes. Esta plataforma permite fazer comparações de diferentes versões de um mesmo arquivo, permissões de ramificações, pesquisa de código-fonte, integração com outros software, inclusive com Jira [11] que é uma ferramenta que permite monitorar o desenvolvimento de tarefas. O Bitbucket foi utilizado neste projeto apenas para armazenar o código-fonte, outros artefatos como documentos base para levantamento de requisitos e documentação foram armazenados no Dropbox. O andamento das tarefas foi acompanhado por meio de sucessivas reuniões.

O Maven é uma ferramenta de automação de compilação utilizada primariamente em projetos Java, mas que também pode ser utilizada para gerenciar projetos desenvolvidos em C#, Ruby ou Scala. O Maven possui um arquivo XML conhecido como POM (*Project Object Model*) que descreve as dependências sobre módulos externos, a ordem de compilação e as propriedades da estrutura do projeto. Ele utiliza dois tipos de repositórios: remoto e local. O remoto fica em um servidor *web* e centraliza todas as bibliotecas que podem ser utilizadas, o local é, basicamente, um diretório na máquina do desenvolvedor onde são encontradas as bibliotecas que este precisa em seus projetos.

[7] https://bitbucket.org/
[8] https://maven.apache.org/
[9] https://subversion.apache.org/
[10] https://www.mercurial-scm.org/
[11] https://br.atlassian.com/software/jira

## 3.5 Avaliação da Abordagem Proposta

A abordagem desenvolvida neste trabalho foi avaliada de maneira qualitativa por meio de uma análise quanto ao atendimento das funcionalidades apontadas por professores em um *survey* e de comparação com os principais trabalhos relacionados encontrados na literatura. Foram definidos critérios de comparação, discutidos no capítulo 6.1 (facilidade de uso, abordagem unificada e grau de flexibilidade), para que um confrontamento justo e objetivo fosse realizado.

O desenvolvimento da pesquisa incluiu sucessivas reuniões com o orientador e alunos de mestrado e graduação. Durante estas reuniões foram traçados o escopo do trabalho e os obstáculos que deveriam ser superados para que a abordagem pudesse ser disponibilizada dentro do cronograma estimado. Uma série de reuniões com o perfil de *brainstorming* e não meramente técnicas foram realizadas com a equipe de desenvolvimento do The Huxley para a integração da abordagem e surgimento de novas ideias.

# 4

# Análise da base do The Huxley

Este capítulo apresenta uma análise dos problemas do The Huxley que podem ser resolvidos por alunos de disciplinas introdutórias de programação. O objetivo é estudar a necessidade dos professores de expressar restrições e se, quando expressas, as mesmas são cumpridas pelos alunos.

O The Huxley possui uma classificação para cada problema, chamada de Nível de Dificuldade (ND). Atualmente, o ND de cada problema pode variar de 1 a 5 (iniciante, fácil, médio, difícil e *expert*). Por conveniência, foi decidido selecionar apenas os problemas com ND de 1 a 3 por se tratarem de problemas mais simples que podem ser resolvidos por alunos que estão aprendendo a programar. O universo dos problemas está restrito aos problemas cadastrados até o final de 2016.

Como a classificação dos problemas é dinâmica, é importante ressaltar que os problemas foram selecionados no dia 19/09/2017. Foram encontrados 173 problemas com ND 4, 173 problemas com ND 5 e 545 problemas com ND abaixo de 4. Dos 545 problemas com ND abaixo de 4, foram removidos da análise 48 problemas que apesar de não terem um ND elevado, eram problemas que exigiam assuntos avançados para que as respostas fossem alcançadas. Alguns desses assuntos são: *backtracking*, programação dinâmica, árvore, pilha, grafos, *insertion sort*, *selection sort* e outros. A lista completa com os 48 problemas e suas respectivas justificativas para remoção está presente no apêndice A.

Diante dos 497 problemas selecionados para análise, foi realizada uma leitura do enunciado de todos eles com o objetivo de descobrir quantos exigiam explicitamente algum tipo de restrição na solução. Foi observado que 76 problemas (15,29%) exigiam que o aluno oferecesse uma solução com algum tipo de restrição. A tabela 4 apresenta os 76 problemas, sua identificação e o tipo de restrição exigida.

Os 76 problemas com restrições coletados nesta análise foram cadastrados por 19 usuários/professores diferentes na plataforma The Huxley. Este dado evidencia que a necessidade de

Tabela 1 – Número de problemas cadastrados por professores

| **Nome** | **Quantidade de problemas** |
|---|---|
| Professor 1 | 28 |
| Professor 2 | 11 |
| Professor 3 | 5 |
| Professor 4 | 5 |
| Professor 5 | 4 |
| Professor 6 | 4 |
| Professor 7 | 3 |
| Professor 8 | 3 |
| Professor 9 | 2 |
| Professor 10 | 2 |
| Professor 11 | 1 |
| Professor 12 | 1 |
| Professor 13 | 1 |
| Professor 14 | 1 |
| Professor 15 | 1 |
| Professor 16 | 1 |
| Professor 17 | 1 |
| Professor 18 | 1 |
| Professor 19 | 1 |

expressar restrições nos enunciados dos problemas não está restrita a um conjunto específico e pequeno de professores. A tabela 1 apresenta a quantidade de problemas cadastrada por cada professor. O professor 1 destaca-se dos demais por ser o usuário padrão do The Huxley que é utilizado pela equipe da plataforma para alimentar a base com problemas obtidos de outras fontes.

Neste ponto, é importante ressaltar que a quantidade de problemas que exigem algum tipo de restrição poderia ser maior se a plataforma The Huxley possuísse algum método de avaliação das soluções. Como os professores e os alunos têm conhecimento que a plataforma não suporta esse recurso, então alguns professores podem ter optado por não fazer esse tipo de exigência. A figura 1 apresenta um resumo de todas as restrições encontradas nos 76 problemas.

A partir da figura 1 é possível perceber que a maior parte das restrições está concentrada em vetor/matriz, função e recursão. Como para implementar uma recursão o aluno também precisa criar uma função (recursiva), é possível somar as ocorrências das restrições de função e recursão para obter um valor generalizado de 32 ocorrências para as restrições de função.

Das restrições encontradas, apenas duas não podem ser especificadas pelo modelo proposto neste trabalho, são elas: *switch* e *do while*. Este trabalho apresenta um modelo de especificação genérico, que atualmente está implementado para as linguagens de programação Java e Python, mas que pode ser estendido facilmente para outras linguagens de programação, como C, C++, C# e Pascal. As restrições *switch* e *do while* não são suportadas porque essas

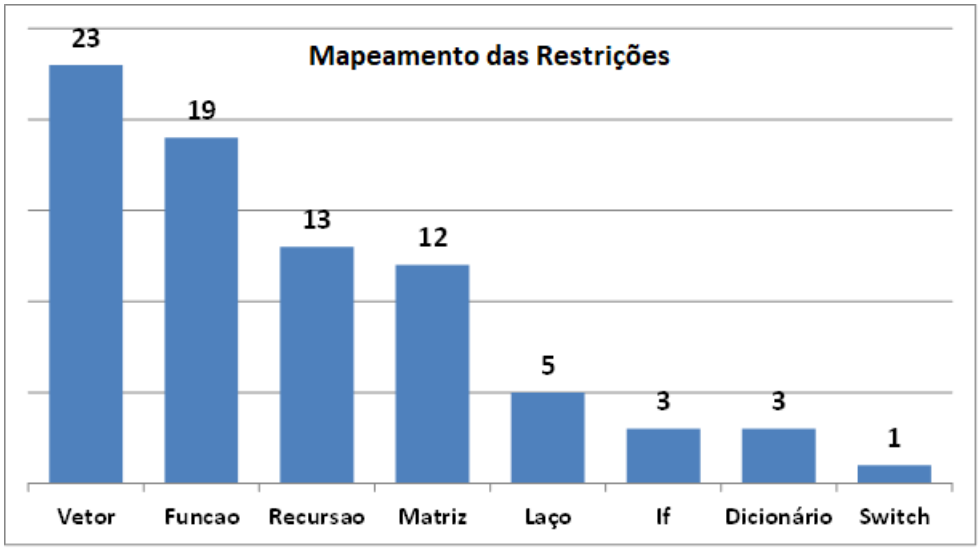


Figura 1 – Restrições encontradas nos 76 problemas

construções não existem na linguagem Python e não seria coerente com a proposta inicial permitir o professor especificar que um problema deve ser resolvido com *do while* se essa construção não existe em todas as linguagens suportadas pelo modelo. O próximo passo após descobrir quais problemas necessitam de algum tipo de restrição é saber se os alunos respeitam as restrições em suas submissões. As seções 4.1 e 4.2 detalham dois problemas com restrições relacionadas à construção *if*.

## 4.1 Decrescente 3

O problema Decrescente 3 (HUXLEY, 2011a) pede que a solução do aluno leia 3 números inteiros e os imprima em ordem decrescente utilizando no máximo 4 comandos IF.

Para analisar se as submissões realmente obedecem às restrições foram selecionadas apenas as submissões que foram consideradas corretas. A figura 2 aponta que 66% das submissões feitas em Java e 25% das submissões feitas em Python que foram consideradas corretas possuem mais do que 4 comandos *if*, portanto não estão de acordo com o que foi solicitado pelo professor.

É importante ressaltar que o problema poderia ser resolvido com até menos do que 4 comandos *if*, mesmo assim muitas submissões não obedeceram à restrição e apesar disso consideradas corretas. As figuras 3 e 4 detalham os erros cometidos pelos alunos neste problema e colaboram com a justificativa da implementação de uma abordagem estática em um sistema de juiz *on-line*.

De acordo com a figura 3, foram aceitas 25 submissões com 5 *ifs* e 14 submissões com 6 *ifs* na linguagem Java. Os casos mais extremos são 1 submissão com 16 *ifs*, 2 submissões com 14 *ifs* e 3 submissões com 13 *ifs*.

De acordo com a figura 4, foram aceitas 45 submissões com 6 *ifs* e 22 submissões com 9

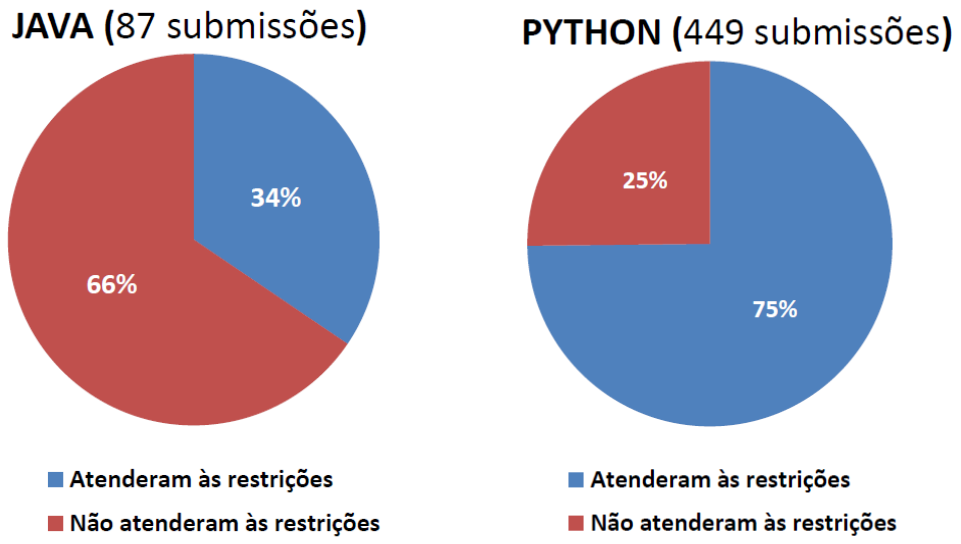


Figura 2 – Resultado da avaliação estática

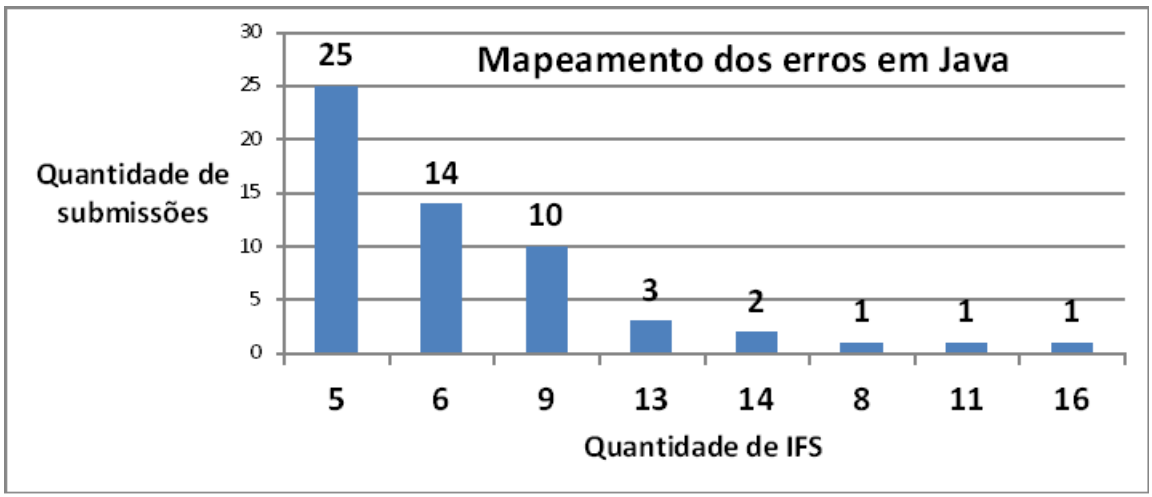


Figura 3 – Mapeamento dos erros em java

*ifs* na linguagem Python. Os casos mais extremos são 4 submissões com 16 *ifs*, 4 submissões com 14 *ifs* e 1 submissão com 18 *ifs*.

## 4.2 O Maior

O problema O Maior (HUXLEY, 2011b) tem como entrada 3 números e pede que o aluno utilize a fórmula 4.1 para descobrir qual o maior entre os 3 números sem utilizar *if*. A intenção do professor é fazer o aluno perceber que a fórmula 4.1 retorna o maior entre 2 números e se for bem utilizada duas vezes retornará o maior entre 3 números sem a necessidade de utilizar

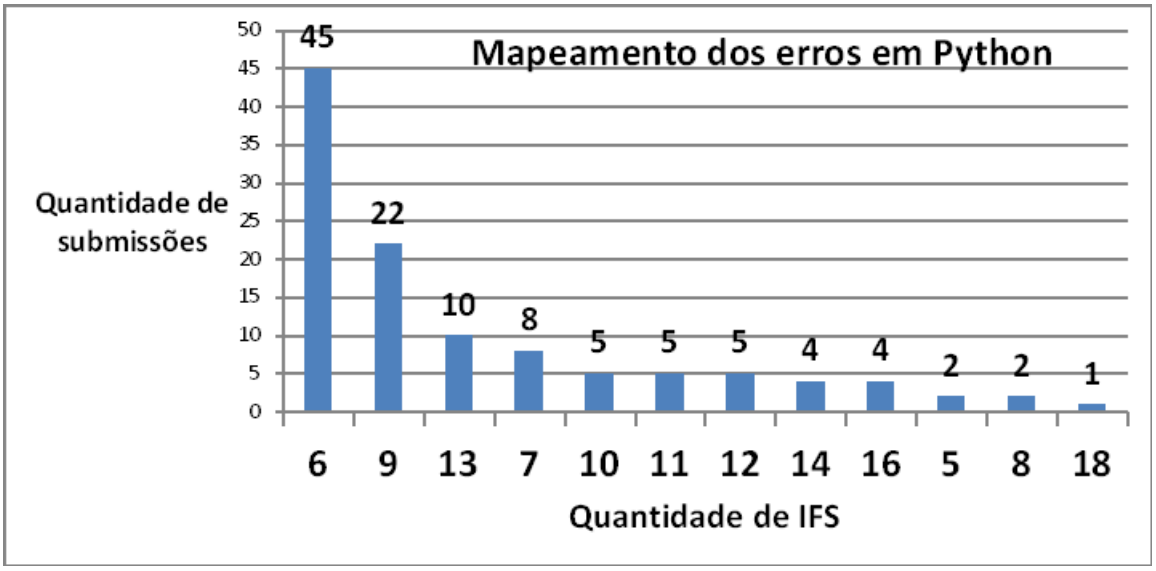


Figura 4 – Mapeamento dos erros em python

o comando *if*.

$$MaiorAB = \frac{a+b+abs(a-b)}{2} \tag{4.1}$$

A figura 5 aponta que 37% das submissões feitas em Java e 14% das submissões feitas em Python foram aceitas mas não atenderam à restrição. As figuras 6 e 7 detalham os erros cometidos pelos alunos.

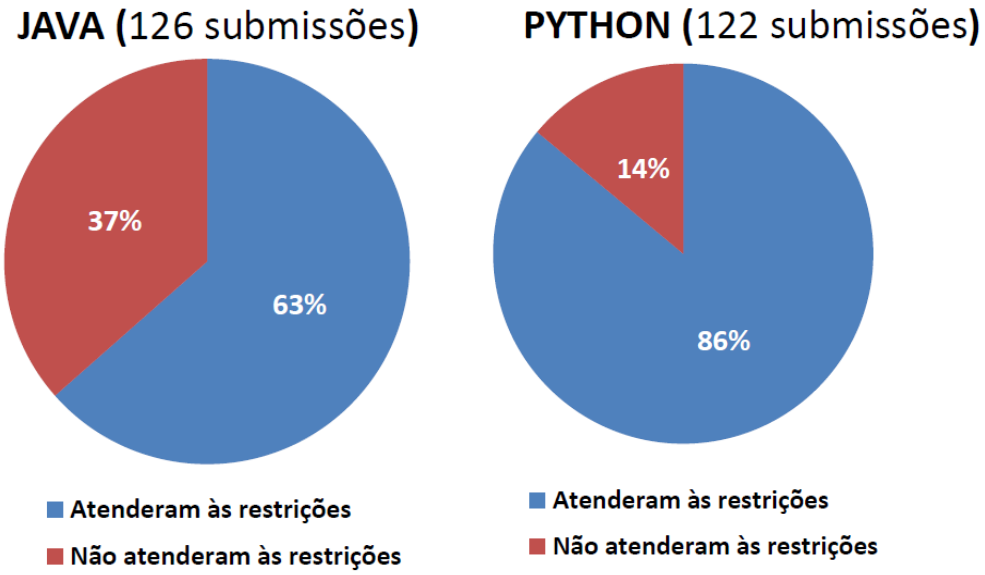


Figura 5 – Resultado da avaliação estática

De acordo com a figura 6, é possível perceber uma concentração de 25 submissões aceitas com 3 comandos *ifs*. Ainda neste problema foram encontradas submissões que utilizaram um

número exagerado de comandos IFS. Para esses casos foram encontradas 2 submissões com 10 *ifs*, 1 submissão com 13 *ifs*, 3 submissões com 14 *ifs* 1 submissão com 16 comandos *ifs*.

De acordo com a figura 7, os casos mais extremos foram de submissões que utilizaram 4 *ifs*. Os casos mais recorrentes foram de submissões que utilizaram 3 *ifs*.

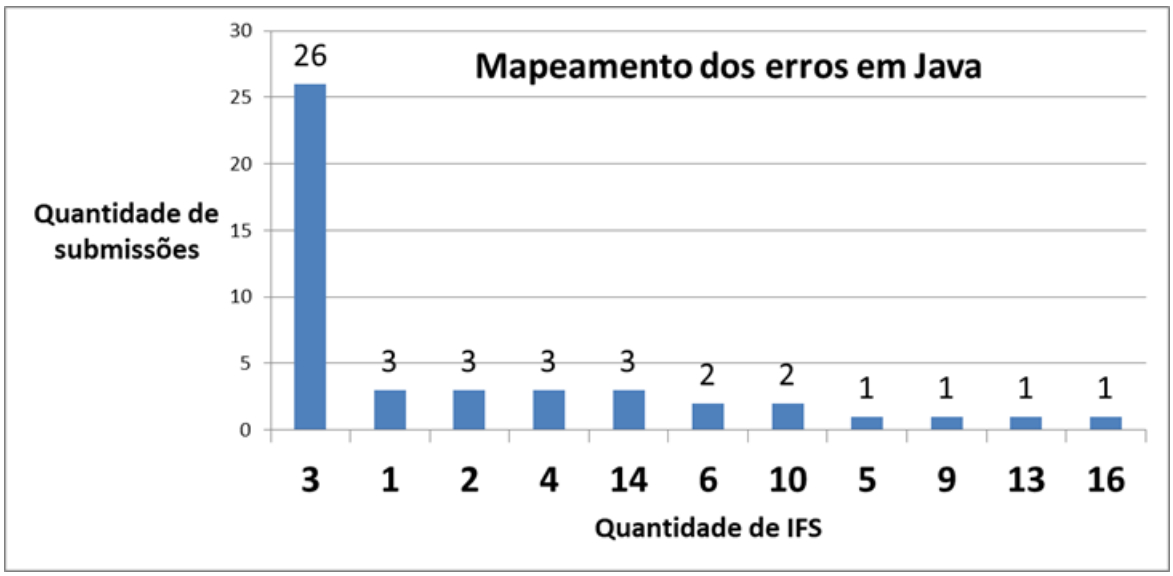


Figura 6 – Mapeamento dos erros em Java

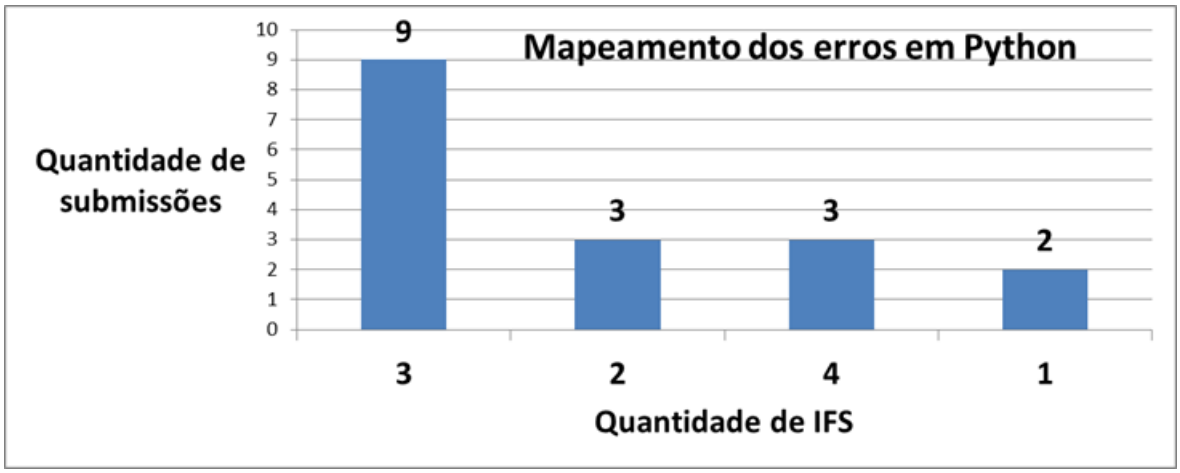


Figura 7 – Mapeamento dos erros em Python

De acordo com os problemas apresentados nas seções 4.1 e 4.2, é notável que apesar da construção *if* ser um comando de fácil compreensão, muitos alunos não conseguem lidar com simples restrições relacionadas a ela. Para evitar esses casos e obrigar o aluno a refletir mais sobre a sua implementação, é indispensável a utilização de uma abordagem que permita checar restrições durante a avaliação das soluções por um juiz *on-line*.

## 4.3 Atendimento às restrições

Nas seções 4.1 e 4.2 foi apresentado o grau de atendimento às restrições para dois problemas específicos. Esta seção discute e apresenta um resumo do grau de atendimento às restrições de todos os problemas selecionados para análise.

A tabela 2 apresenta um resumo do grau de atendimento classificado por restrição. A primeira coluna possui o nome da restrição, a segunda possui a quantidade de submissões consideradas corretas pela plataforma The Huxley, a terceira possui quantas das submissões corretas realmente atenderam às restrições estabalecidas e finalmente a quarta coluna apresenta um percentual de atendimento às restrições.

Tabela 2 – Resumo do atendimento às restrições

| **Restrição** | **Submissões** | **Atenderam à restrição** | **Percentual** |
|---|---|---|---|
| Array/Matriz | 3138 | 3089 | 98% |
| Repetição | 147 | 114 | 78% |
| Dicionário | 101 | 78 | 77% |
| IF | 798 | 564 | 71% |
| Função | 2972 | 1523 | 51% |
| Função Geral | 5926 | 2463 | 47% |
| Recursão | 2230 | 940 | 42% |

De acordo com a tabela 2, a restrição menos atendida pelo alunos foi implementar uma função recursiva com um determinado nome e quantidade de parâmetros. É notável que essa restrição é a mais fácil de não ser atendida porque o problema pode ser resolvido com outras alternativas. O aluno pode utilizar um laço de repetição (*for*, *while* ou *do while*) no lugar da recursão e os testes caixa-preta não identificariam erros. Outra maneira de não seguir a restrição é sequer criar uma função e não praticar a capacidade de modularizar os programas.

O caso da Recursão se torna ainda mais interessante analisando 4 problemas específicos que exigiram essa restrição como está apresentado na tabela 3.

As duas últimas colunas da tabela 3 apresentam o percentual de submissões que realmente foram submetidas atendendo à restrição de recursão.

Tabela 3 – Os piores casos de atendimento à restrição de recursão

| **Nome dos problemas** | **Submissões em Python** | **Submissões em Java** | **Atendimento às restrições em Python** | **Atendimento às restrições em Java** |
|---|---|---|---|---|
| Contando os Dígitos Pares | 50 | 156 | 16% | 19% |
| Série de Miguelito | 63 | 200 | 40% | 10% |
| Sequência misteriosa | 41 | 2 | 2% | 0% |
| Sequência de inteiros pares (crescente) recursivo | 250 | 227 | 16% | 16% |

Ainda de acordo com a tabela 2, é possível perceber que a restrição de Array/Matriz quase sempre é obedecida pelos alunos com um percentual de 98%. Neste ponto é importante ressaltar como foi realizada a correção estática das submissões. A linguagem Java possui os conceitos de Array/Matriz de forma explícita, entretanto não existe na linguagem Python. Para esta linguagem foi considerado que quando a submissão utilizasse o conceito de lista, o objetivo da restrição estava alcançado. Essa observação é importante porque em Java o conceito de lista está relacionado às classes *ArrayList* (ORACLE, 1998a), *LinkedList* (ORACLE, 1998b) ou *Vector* (ORACLE, 1996) e não à declaração explícita de um vetor/matriz.

As restrições de IF, Repetição e Dicionário possuem um conjunto mais restrito de submissões e um grau de atendimento maior do que as restrições Função e Recursão. A penúltima linha da tabela 2 apresenta os dados de Função e Recursão agrupados como Função Geral.

A tabela 4 apresenta com detalhes todos os problemas analisados neste trabalho. A primeira coluna possui o nome do problema, a segunda o tipo de restrição estabelecida, a terceira a quantidade de submissões em Python, a quarta a quantidade de submissões em Java, a quinta o percentual de atendimento às restrições em Python e a última o mesmo percentual em Java.

Tabela 4 – Problemas com restrições

| **Nome do problema** | **Restrição** | **Submissões em Python** | **Submissões em Java** | **Atendimento às restrições em Python** | **Atendimento às restrições em Java** |
|---|---|---|---|---|---|
| Estações do Ano | função | 293 | 9 | 42% | 0% |
| Menor velocidade final de 3 carros | função | 256 | 4 | 57% | 0% |
| Média Ponderada de 4 notas | função | 116 | 9 | 43% | 44% |
| Inverter uma Lista Encadeada em uma só passada | função | 89 | 224 | 37% | 93% |
| Decrescente 3 | *if* | 449 | 87 | 75% | 34% |
| Calcular MDC de forma recursiva | recursão | 61 | 45 | 52% | 82% |
| Transformar Decimal para Binário (recursivo) | recursão | 106 | 29 | 34% | 55% |
| Contando os Dígitos Pares | recursão | 50 | 156 | 16% | 19% |
| Série de Miguelito | recursão | 63 | 200 | 40% | 10% |

| | | | | | |
|---|---|---|---|---|---|
| Base e expoente. | repeticao | 9 | 20 | 22% | 25% |
| O Maior | *if* | 122 | 126 | 86% | 63% |
| Matrizes | vetor/matriz | 0 | 101 | - | 100% |
| Inverso | vetor/matriz | 121 | 187 | 100% | 99% |
| Iguais a n | vetor/matriz | 238 | 36 | 100% | 97% |
| Multiplicação da Diagonal de uma Matriz | vetor/matriz | 52 | 165 | 100% | 100% |
| Fábrica de motores | vetor/matriz | 8 | 2 | 100% | 100% |
| Brincando com Arrays | vetor/matriz | 121 | 159 | 100% | 97% |
| Matriz caracol | vetor/matriz | 6 | 9 | 100% | 100% |
| Arrays de pares e ímpares | vetor/matriz | 24 | 7 | 100% | 100% |
| Intercalar um array | vetor/matriz | 358 | 188 | 100% | 99% |
| Interseção entre listas | vetor/matriz | 112 | 14 | 99% | 64% |
| Linha da Matriz | vetor/matriz | 40 | 162 | 100% | 99% |
| Menor Valor e Posição | vetor/matriz | 297 | 181 | 100% | 97% |
| Matriz Quadrada I | vetor/matriz | 10 | 161 | 100% | 99% |
| Matriz escada | vetor/matriz | 1 | 6 | 100% | 100% |
| Colisão no Mapa | vetor/matriz | 6 | 3 | 83% | 100% |
| Rotacionar um array | vetor/matriz | 1 | 0 | 100% | - |
| Quais os números que estão faltando? | vetor/matriz | 68 | 6 | 99% | 83% |
| Ordenação de matrizes | vetor/matriz | 5 | 8 | 100% | 100% |
| Multiplicação de Matrizes | vetor/matriz | 17 | 7 | 100% | 100% |
| Soma números grandes utilizando arrays | vetor/matriz | 8 | 51 | 75% | 94% |

| | | | | | |
|---|---|---|---|---|---|
| Acessando notas | vetor/matriz | 0 | 1 | - | 100% |
| Intersecção de arrays | vetor/matriz | 5 | 1 | 100% | 100% |
| Elementos de uma matriz | vetor/matriz | 18 | 6 | 100% | 100% |
| Valores digitados. | vetor/matriz | 6 | 19 | 50% | 89% |
| Ordem Inversa. | vetor/matriz | 2 | 17 | 100% | 100% |
| Posições ímpares. | vetor/matriz | 2 | 15 | 100% | 87% |
| Quadrado da posição. | vetor/matriz | 3 | 18 | 67% | 72% |
| valor do array | vetor/matriz | 0 | 0 | - | - |
| Idade e altura. | vetor/matriz | 0 | 0 | - | - |
| pedagio do mês | vetor/matriz | 0 | 0 | - | - |
| Média do produto. | vetor/matriz | 0 | 0 | - | - |
| Loja de sapatos. | vetor/matriz | 0 | 2 | - | 100% |
| o melhor sistema operacional | vetor/matriz | 0 | 0 | - | - |
| Substitui no array | vetor/matriz | 10 | 67 | 90% | 99% |
| Piscamos tanto assim? | vetor/matriz | 0 | 0 | - | - |
| Pesquisa de intenção de voto | função | 182 | 4 | 46% | 0% |
| Ordenar Lista Encadeada em ordem crescente | função | 21 | 166 | 24% | 91% |
| Conteúdo de Duas Listas Encadeadas | função | 79 | 182 | 46% | 87% |
| Frequência de um texto em uma Lista Encadeada | função | 46 | 3 | 33% | 67% |
| Somas sucessivas como multiplicação | recursão | 303 | 212 | 84% | 84% |
| Sequência de inteiros pares (crescente) recursivo | recursão | 250 | 227 | 16% | 16% |

| | | | | | |
|---|---|---|---|---|---|
| Sequência de inteiros pares (decrescente) recursivo | recursão | 217 | 52 | 31% | 62% |
| Hospedagem na Praia | função | 54 | 3 | 57% | 33% |
| Limite de Velocidade | função | 486 | 11 | 47% | 27% |
| Soma dos Dígitos | função | 186 | 11 | 11% | 18% |
| Escola de Música | função | 246 | 7 | 35% | 0% |
| Algoritmo de Euclides | recursão | 46 | 0 | 57% | - |
| Restaurante Universitário | *if* | 10 | 4 | 100% | 75% |
| Contagem - Antecessor é Maior | repeticao | 0 | 0 | - | - |
| O dobro do maior | função | 27 | 28 | 7% | 11% |
| Períodos da História | função | 15 | 5 | 33% | 0% |
| Sequência misteriosa | recursão | 41 | 2 | 2% | 0% |
| Moda de Conjunto de Inteiros | dicionario | 57 | 1 | 95% | 0% |
| Lista Ordenada | recursão | 15 | 101 | 0% | 37% |
| Períodos da História Numéricos | função | 6 | 6 | 67% | 17% |
| Análise de Molde de DNA | dicionario | 19 | 9 | 47% | 11% |
| Exponenciação rápida | recursão | 15 | 133 | 0% | 70% |
| Vogais no texto | dicionario | 43 | 0 | 56% | - |
| Lista de Produtos | função | 0 | 40 | - | 100% |
| Array está ordenado-Recursivo | recursão | 22 | 0 | 41% | - |
| Volume da Esfera | função | 137 | 12 | 60% | 0% |
| Dois maiores entre 10 | repeticao | 1 | 70 | 0% | 86% |

| | | | | | |
|---|---|---|---|---|---|
| Cara ou coroa | recursão | 0 | 0 | - | - |
| Tabela de vendas | repeticao | 2 | 45 | 100% | 100% |
| Formatando número real como moeda | função | 2 | 7 | 50% | 0% |

# 5

# Abordagem Proposta

Este capítulo apresenta em detalhes a forma como nossa abordagem está organizada e as decisões de projeto que foram tomadas para sua realização.

A proposta deste trabalho é oferecer uma abordagem para especificação e checagem de restrições (requisitos) relacionadas às construções de várias linguagens de programação. Esta abordagem deve ser fácil de ser utilizada pelo professor (usuário), não necessitando de que o mesmo possua conhecimento específico sobre a formação da AST de uma solução nem precise programar para realizar as especificações. Além disso, esta abordagem deve levar em consideração os aspectos comuns das principais linguagens de programação estruturais e orientadas a objetos.

## 5.1 *Design* da solução

A implementação da abordagem neste trabalho foi projetada na linguagem Java e teve o objetivo de servir como api/biblioteca para um sistema de juiz *on-line*. Apesar disso, nada impede que futuramente ela seja disponibilizada em uma arquitetura de *web-services* para manter a compatibilidade com sistemas não implementados em Java.

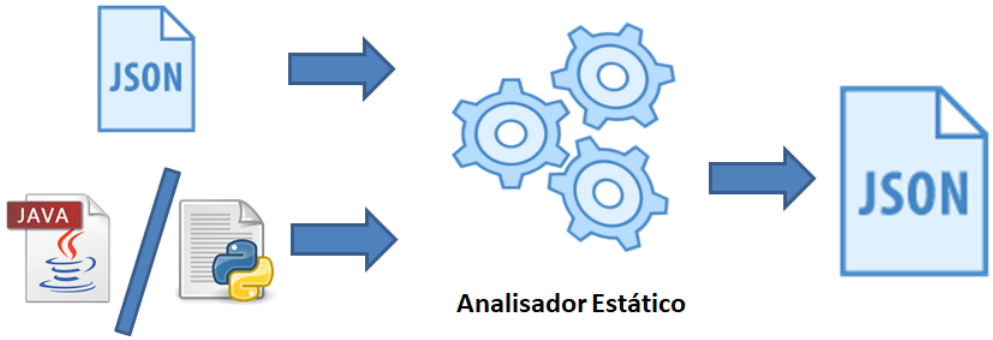


Figura 8 – Arquitetura da solução

A figura 8 apresenta de maneira geral o funcionamento da nossa abordagem. No núcleo temos o Analisador Estático que foi implementado em Java com auxílio do *framework* ANTLR 4 (PARR, 2013) e com as gramáticas da linguagem Java 7 (DWARS, 2013) e Python 3 (EVERETT, 2014).

O Analisador Estático tem como entrada o arquivo de especificação de restrições no formado *Json*, o código-fonte com a solução do aluno e a linguagem de programação da solução. A saída do Analisador Estático é um arquivo *Json* que informa se a solução está de acordo com as restrições e se não estiver também informa os problemas encontrados.

### 5.1.1 Exemplo de utilização

Esta subseção apresenta de maneira geral como esta abordagem pode ser utilizada por outros sistemas. Os delalhes de como devem ser construídos e interpretados todos os tipos de arquivos *Json* de entrada e saída estão no apêndice B.

Inicialmente, o cliente (juiz *on-line*) deve criar um arquivo *Json* com todas as restrições que deseja que sejam analisadas nas soluções dos alunos. A listagem 5.1 apresenta um exemplo de uma restrição que exige a criação e utilização de uma função recursiva chamada fibonacci com retorno e apenas um parâmetro. Além disso, há uma exigência de que a solução tenha exatamente 2 desvios condicionais.

```
{
  "restricaoFuncoes":[
    {"nome":"fibonacci",
    "retorno":true,
    "recursiva":true,
    "qtdParametros":1}],

  "restricaoIf":{
    "minimo":2,
    "maximo":2}
}
```

Listagem 5.1 – Exemplo de uma representação de restrições.

O segundo arquivo de entrada para o analisador estático é a solução propriamente dita. A figura 5.1 apresenta uma possível solução para este problema na linguagem Java.

```
package codigo.exemplo.dissertacao;
public class ExemploFibonacci {
    public static int fibonacci(int numero) {
        if (numero < 2) {
            return numero;
        } else {
            return fibonacci(numero - 1) + fibonacci(numero - 2);
```

```
8           }
9       }
10      public static void main(String[] args) {
11          int numero = 5;
12          // Imprime o 5 (quinto) numero da sequencia de fibonacci
13          System.out.println(fibonacci(numero));
14      }
15  }
```

Listagem 5.2 – Solução que atende às restrições da listagem 5.1

Depois de o cliente conseguir os dois arquivos (*Json* de restrições e solução do aluno), o mesmo deve se comunicar com a API para obter a resposta do analisador estático. Essa comunicação é feita como mostra a listagem 5.3

```
1  package exemplo.codigo.dissertacao;
2  import java.io.File;
3  import java.io.IOException;
4  import java.nio.file.Files;
5  import analisador.estatico.base.Constantes;
6  import analisador.estatico.ufs.AnalisadorEstatico;
7  import analisador.estatico.ufs.ResultadoAnalise;
8  public class ComunicacaoAPI {
9   public static void main(String args[]) throws Exception {
10   String solucaoJava = "";
11   String restricaoJson = "";
12   try {
13    //Leitura do arquivo das restricoes
14    File json = new File("restricoes.json");
15    restricaoJson = new String (Files.readAllBytes(json.toPath()));
16    //Leitura do arquivo da solucao
17    File solucao = new File("solucao.java");
18    solucaoJava = new String (Files.readAllBytes(solucao.toPath()));
19    } catch (IOException e) {
20     System.err.printf("Erro na abertura do arquivo: %s.\n", e.
      getMessage());
21    }
22
23   AnalisadorEstatico analisador = new AnalisadorEstatico(Constantes.
      JAVA, solucaoJava, restricaoJson);
24   ResultadoAnalise resultado =  analisador.analisar();
25   // Retorna uma string com o detalhes da analise
26   String resultadoAnalise = resultado.getRetornoJson();
```

```
27     }
28 }
```

Listagem 5.3 – Exemplo de comunicação com a API 5.1

As primeiras linhas da listagem 5.3 são apenas para obter os arquivos de restrição e da solução, portanto podem variar de acordo com a plataforma que está utilizando a API. Na linha 23 é feita a comunicação com o analisador estático, para instanciação do objeto AnalisadorEstatico é necessário informar o tipo de linguagem de programação da solução, a solução e as restrições. Na linha 24 é acionado o processamento do analisador estático e na linha 26 é retornada a avaliação feita pelo mesmo. No exemplo em questão, o resultado da avaliação é apresentado na listagem 5.4.

```
1 {
2   "retornoFuncoes":
3     [{"existe":true,
4     "nome":true,
5     "parametros":true,
6     "recursao":true,
7     "retorno":true,
8     "foiChamada":true}],
9
10  "retornoIf":
11    {"atendeu":true,
12    "quantidade":2}
13 }
```

Listagem 5.4 – Resultado da análise.

Como a solução utilizada para este exemplo estava de acordo com as restrições, a avaliação foi completamente positiva. Neste ponto, é importante perceber o nível de detalhe da avaliação. Para o caso da função recursiva são realizadas verificações em casacata, nesta ordem:

- Existe função?
- A função possui o nome esperado?
- A função possui a quantidade de parâmetros esperada?
- Atende aos requisitos de recursão e retorno?
- Foi invocada?

Este nível de detalhe é importante porque pode ser utilizado pelo juiz *on-line* para dar informações mais precisas ao aluno a respeito do seu erro.

No caso da avaliação dos desvios condicionais, o analisador retorna a quantidade de *ifs* encontrados na solução.

### 5.1.2 Padrão *Abstract Factory*

O padrão de projeto *Abstract Factory* fornece uma interface para criação de família de objetos relacionados ou dependentes sem especificar suas classes concretas. (ERICH et al., 2000)

Este padrão deve ser usado em situações quando:

- Um sistema deve ser independente de como seus produtos são criados, compostos ou representados; (ERICH et al., 2000)
- Deseja-se fornecer uma biblioteca de classes de produtos e apenas revelar suas interfaces, não suas implementações; (ERICH et al., 2000)
- Um sistema deve ser configurado como um produto de uma família de múltiplos produtos. (ERICH et al., 2000)

Os motivos para utilização do padrão criacional *Abstract Factory* estão alinhados com o propósito deste trabalho. Nesta pequisa foi desenvolvida uma biblioteca que possui uma família de múltiplos produtos (algoritmos) com propósitos iguais, mas implementações diferentes. Desta forma, é desejável que o cliente apenas se comunique com as interfaces e não com as classes concretas.

A figura 9 apresenta a estrutura geral do *Abstract Factory*. A partir dela é possível perceber que o cliente se comunica apenas com as classes abstratas que possuem um contrato que deve ser seguido pelas classes concretas.

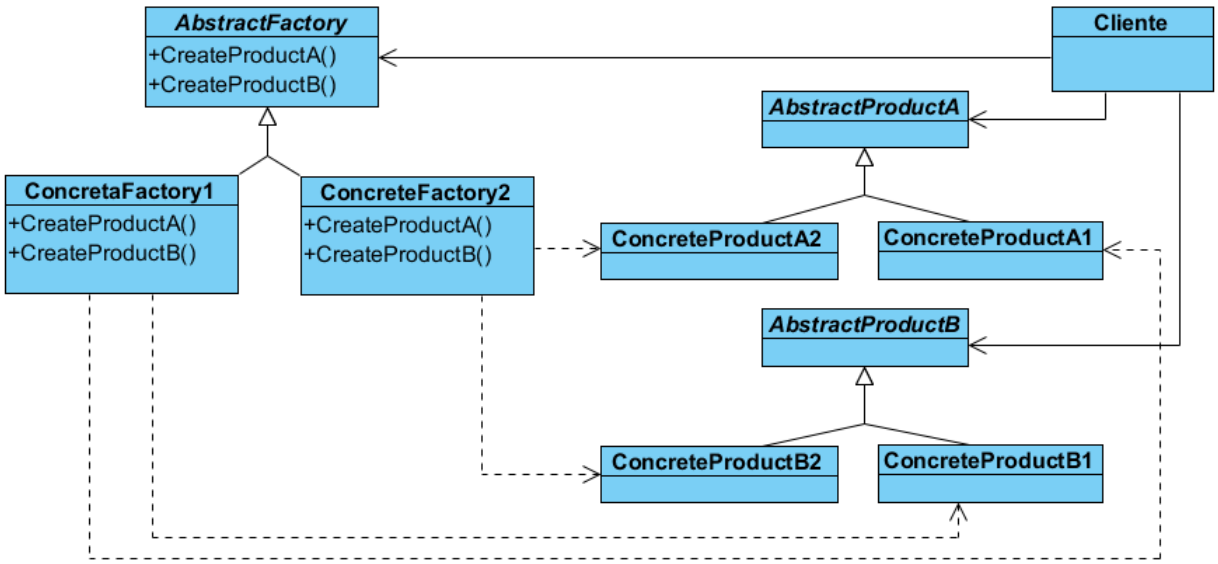


Figura 9 – Diagrama de classes do *Abstract Factory*

A figura 10 apresenta como este padrão de projeto foi aplicado neste trabalho. Dessa forma, percebe-se que se futuramente for necessário adicionar outras linguagens de programação,

as classes abstratas não serão alteradas, portanto a comunicação com o cliente sofrerá apenas pequenas alterações que se limitam a informar qual a linguagem de programação utilizada.

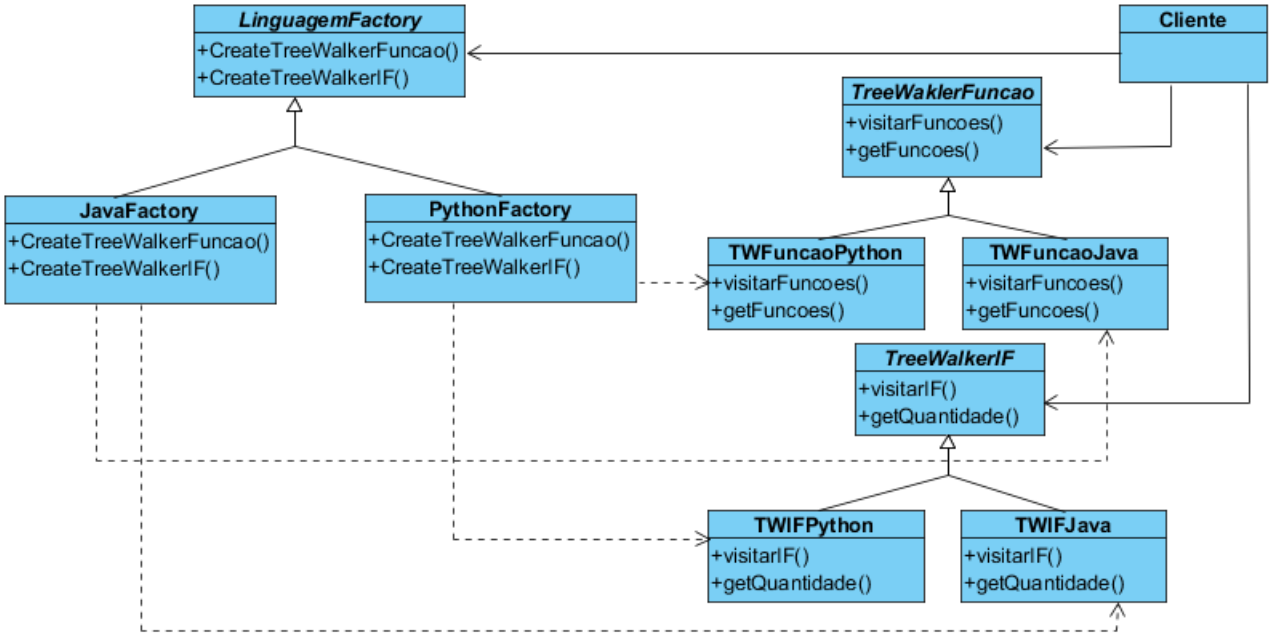


Figura 10 – *Abstract Factory* adaptado para nossa abordagem

Para uma melhor compreensão da aplicação deste padrão de projeto, a figura 11 apresenta uma AST referente à listagem 5.5 que corresponde a uma código Java e a figura 12 apresenta uma AST referente à listagem 5.6 que corresponde a um código Python. Apesar de serem representações de duas linguagens de programação diferentes, elas apresentam a mesma semântica: criação de uma função não recursiva com dois parâmetros e com retorno. Para simplificar a apresentação das ASTs, as listagens não possuem chamadas das funções criadas. Entretanto, é importante ressaltar que sempre que o professor especificar que uma função deve ser criada a mesma também deverá ser chamada.

```
1 public int somar(num1, num2){
2     return num1 + num2;
3 }
```

Listagem 5.5 – Código da AST da figura 11

```
4 def somar(num1, num2):
5     return num1 + num2
```

Listagem 5.6 – Código da AST da figura 12

A classe abstrata *TreeWalkerFuncao* da figura 10 possui a operação *getFuncoes()* que retorna todas as funções encontradas no código-fonte. Neste ponto, é importante perceber que o cliente se comunica com esta classe abstrata e não com as classe concretas, portanto para ele

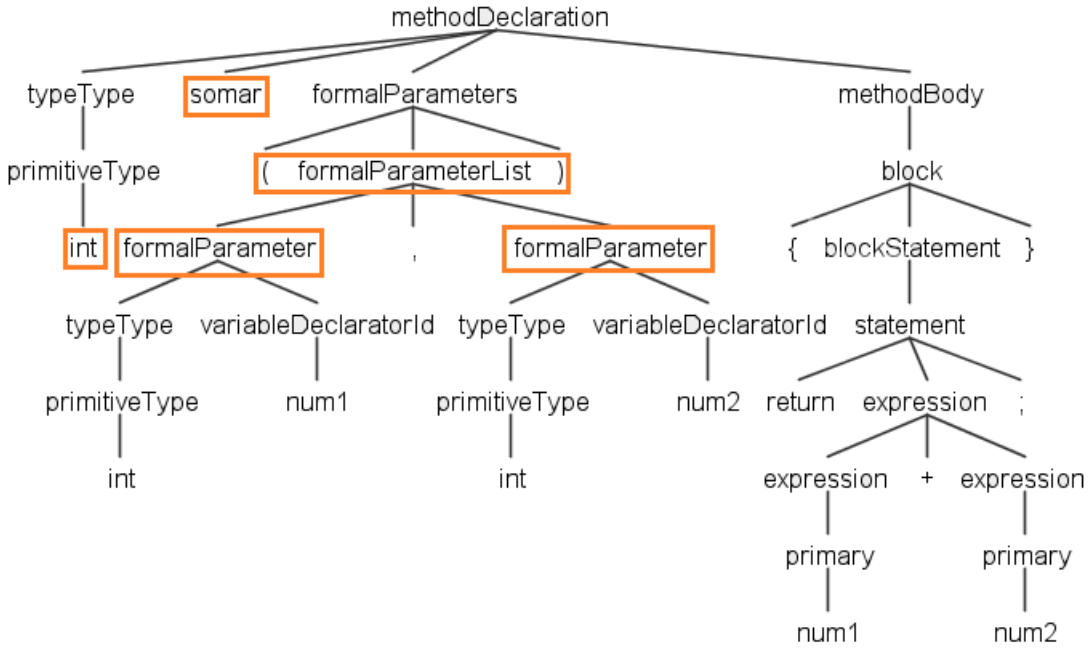


Figura 11 – Árvore Sintática Abstrata de código-fonte Java

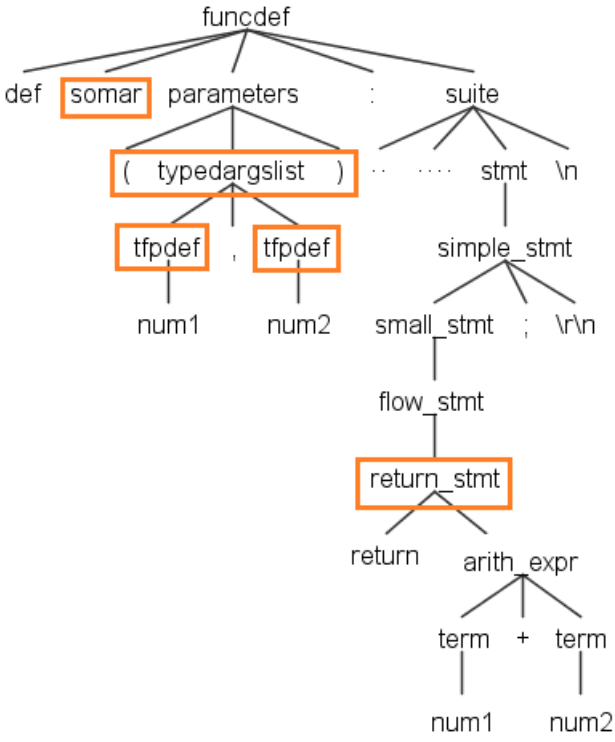


Figura 12 – Árvore Sintática Abstrata de código-fonte Python

é abstraída a ideia de que existem diversas implementações que analisam AST's de diferentes linguagens.

Essa característica possibilitada pela aplicação do padrão de projeto permite que novos algoritmos que analisam AST's de outras linguagens de programação fora do escopo deste

trabalho (como C, C++, C# ou Pascal) sejam adicionados sem que o cliente precise realizar grande alterações.

## 5.2 O Projeto

O projeto foi desenvolvido com as ferramentas Eclipse [1], o sistema de controle de versão distribuído Git [2] e o repositório de projetos Bitbucket [3]. Atualmente ele está estruturado em 10 pacotes e 72 classes. A figura 13 apresenta todos os pacotes do sistema.

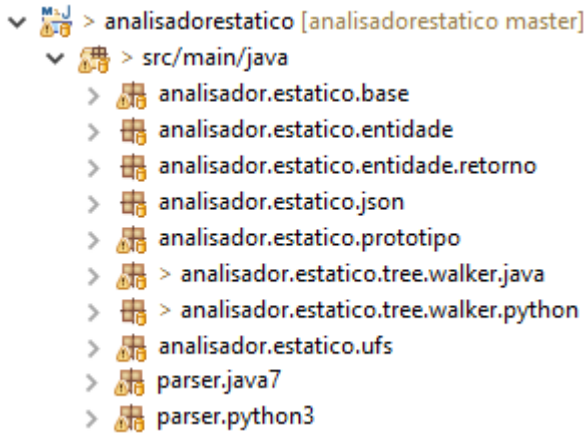


Figura 13 – Pacotes do projeto

O pacote *analisador.estatico.base* possui em sua maior parte Interfaces que são a base do *Abstract Factory*. Elas definem as operações que devem ser obrigatoriamente executadas pelas classes concretas dos pacotes *analisador.estatico.tree.walker.java* e *analisador.estatico.tree .walker.python*. Esses dois pacotes possuem as implementações específicas de cada linguagem de programação. As figuras 14 e 15 mostram as classes desses pacotes.

O pacote *analisador.estatico.entidade* possui classes concretas simples, apenas com atributos e operações *getters* e *setters* para representar as restrições que são obtidas por meio do arquivo *Json* de entrada. Este pacote pode ser utilizado pelo cliente/Juiz *on-line* para auxílio na criação dos arquivos *Json*. A figura 16 apresenta as classes deste pacote.

O pacote *analisador.estatico.entidade.retorno* possui classes semelhantes às classes do pacote anterior, entretanto para representar as informações do arquivo *Json* de saída. A figura 17 possui as classes desse pacote.

O pacote *analisador.estatico.ufs* possui classes que se comunicam com as Interfaces definidas no pacote base. Além disso, ele possui a interface de comunicação com o The Huxley e pode ser entendido como uma fachada. Os pacotes *parser.java7* e *parser.python3* possuem as classes geradas pelo ANTLR 4. As classes desses pacotes são mostradas na figura 18

[1] http://www.eclipse.org/
[2] https://git-scm.com/
[3] https://bitbucket.org/

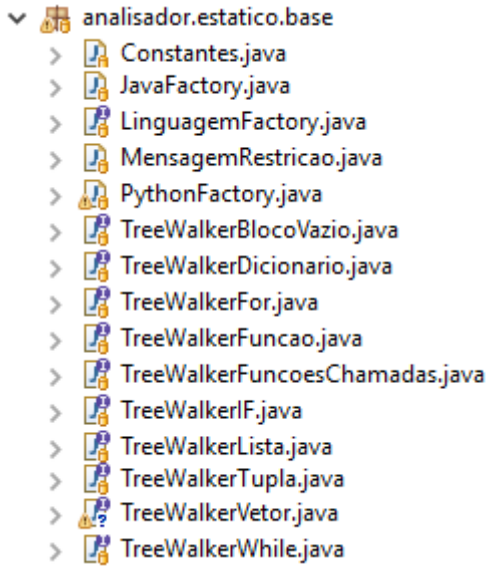


Figura 14 – Pacote *analisador.estatico.base*

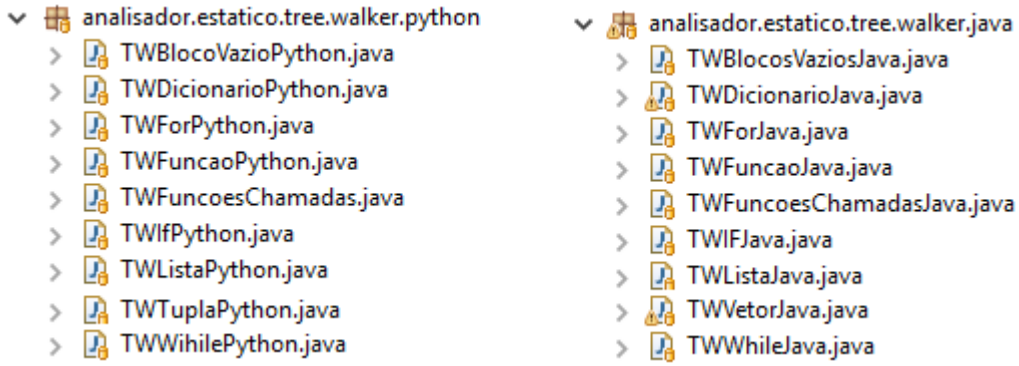


Figura 15 – Pacote com classes concretas dos *Tree Walkers*

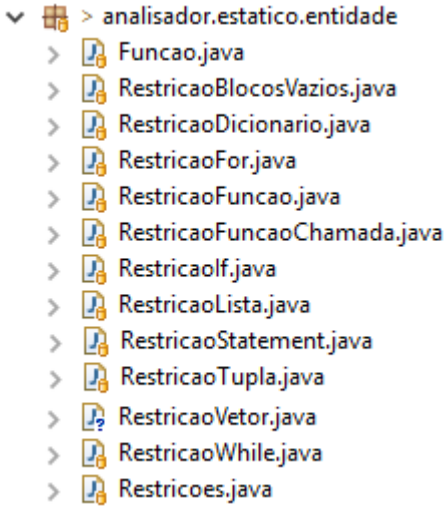


Figura 16 – Pacote *analisador.estatico.entidade*

O pacote *analisador.estatico.json* possui apenas uma classe responsável por manipular os arquivos *Json*. O pacote *analisador.estatico.prototipo* possui as classes responsáveis pelos testes

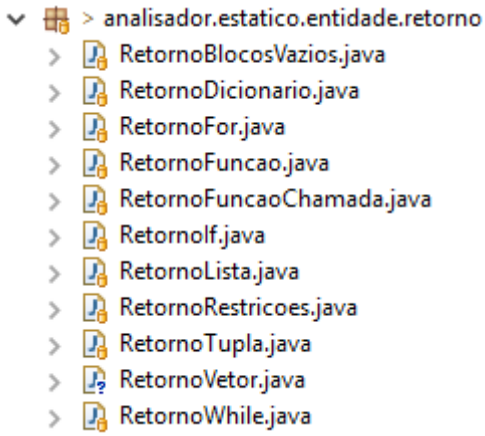


Figura 17 – Pacote *analisador.estatico.entidade.retorno*

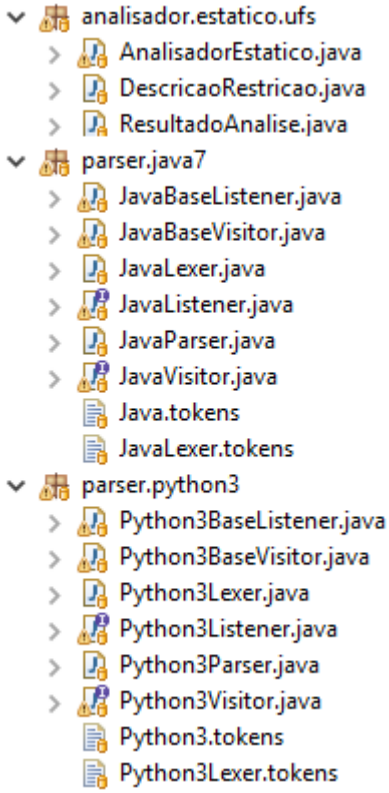


Figura 18 – Pacote *analisador.estatico.ufs* e *parsers*

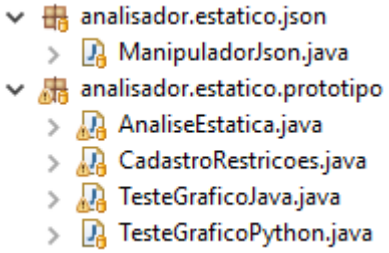


Figura 19 – Pacote *analisador.estatico.ufs* e *analisador.estatico.json*

realizados durante todo o desenvolvimento. As classes *TesteGraficoJava* e *TesteGraficoPython* mostram a Árvore Sintática Abstrata do código-fonte que está sendo testado. O desenvolvimento de Tree Walkers é uma tarefa que exige a execução frequente de testes de unidade devido à grande variedade de cenários possíveis. Por exemplo: Há diversas formas de declarar um vetor/matriz em Java e todas elas devem ser testadas, um aluno pode criar uma função e não chamá-la (isso deve ser notificado) entre outras situações que envolvem outros *Tree Walkers*. A figura 19 possui as classes desses pacotes.

## 5.3 Grau de flexibilidade

O grau de flexibilidade desta proposta está alinhado com o que foi percebido no levantamento apresentado no capítulo 4 e com o resultado do *survey* apresentado no capítulo 6.1.4.3.

Para que os professores não precisem programar e apenas interajam com uma interface gráfica para especificar suas restrições, é necessário implementar algoritmos pré-definidos que realizem os caminhamentos nas ASTs das soluções. Esses algoritmos são chamados de *Tree Walkers*. Nesta pesquisa foram desenvolvidos os *Tree Walkers* apresentados no quadro 5.1.

| **Tree Walkers** | **Subseção** | **Descrição** |
|---|---|---|
| Desvios condicionais | 5.3.1 | Conta quantos desvios condicionais *if* a solução possui. |
| *While* e *For* | 5.3.2 | Conta quantos *while* e *for* a solução possui. |
| Função | 5.3.3 | Verifica se uma função foi criada e invocada. |
| Blocos Vazios | 5.3.4 | Verifica se há blocos vazios. |
| Array/Matriz | 5.3.5 | Verifica se a solução possui declaração de Array/Matriz. |
| Lista e Dicionário | 5.3.6 | Verifica se a solução possui declaração de lista ou dicionário. |
| Chamada de função | 5.3.7 | Verifica se uma função foi invocada. |

Quadro 5.1 – Tree Walkers implementados

Todos os *Tree Walkers* apresentados nesta seção foram desenvolvidos para soluções Python e Java com o objetivo de demonstrar que esta abordagem funciona para linguagens com características diferentes. As próximas subseções detalham o funcionamento de cada *Tree Walker*. Os arquivos *Json* que especificam as restrições e são entradas para o analisador estático estão presentes no apêndice B.

### 5.3.1 Desvios Condicionais

Este *Tree Walker* permite que o professor especifique quantos desvios condicionais a solução do aluno deve ter. Essa especificação é realizada em faixas por meio de um simples

preenchimento de dois campos gráficos indicando o valor mínimo e o valor máximo de desvios condicionais que a solução deve ter. As listagens 5.7 e 5.8 apresentam dois códigos-fonte com 3 desvios condicionais (*if, elif, else*). É importante destacar que este *Tree Walker* não verifica outras construções condicionais como *switch/case*.

```
entrada = int(raw_input())
if(entrada == 0):
  print ("Zero")
elif(entrada > 0):
  print ("Positivo")
else:
  print ("Negativo")
```

Listagem 5.7 – Solução com 3 desvios condicionais em Python

```
package codigo.exemplo.dissertacao;
public class ExemploIF {
    public void positivoNegativo(int numero) {
        if(numero == 0)
            System.out.println("Zero");
        if(numero > 0)
            System.out.println("Positivo");
        else
            System.out.println("Negativo");
    }
}
```

Listagem 5.8 – Solução com 3 desvios condicionais em Java

### 5.3.2 While e For

Neste caso é permitido que o professor especifique a quantidade de *while* e *for* que a solução deve ter. Assim como apresentado na subseção anterior, neste caso também é possível especificar uma faixa de valores visando oferecer uma maior flexibilidade ao professor. As listagens 5.9 e 5.10 apresentam dois códigos-fonte que utilizam um laço *for* para identificar o maior número de uma lista. Para resolver um problema desse tipo o professor poderia especificar que ele deveria ser resolvido com um laço (*for* ou *while*) e apenas 1 desvio condicional *if*.

```
lista = (10,1,2,5,6,19)
maior = lista[0]
for i in lista:
    if maior < i:
        maior = i
print("O maior eh ", maior)
```

Listagem 5.9 – Solução com 1 laço for em Python

```java
package codigo.exemplo.dissertacao;
import java.util.ArrayList;
public class ExemploRepeticao {
    public void OMaior(ArrayList<Integer> lista) {
        int maior = lista.get(0);
        for(int i = 1; i < lista.size(); i++) {
            if(maior < lista.get(i))
                maior = lista.get(i);
        }
        System.out.println("O maior eh "+maior);
    }
}
```

Listagem 5.10 – Solução com 1 laço for em Java

Vale ressaltar que o analisador estático não diferencia construções aninhadas. Portanto, se o professor especificar que a solução precisa de um *for* e um *while*, esses podem estar aninhados ou não. Essa característica visa não prender o aluno a um formato específico de solução, pois o objetivo do trabalho é fazer o aluno aprender a utilizar as construções da linguagem corretamente.

### 5.3.3 Função

Conforme apresentado no capítulo 4, há uma necessidade grande de forçar o aluno a modularizar suas soluções. Este *Tree Walker* serve para que o professor possa especificar que a solução deve criar e utilizar uma função informando:

- Um determinado nome
- Uma quantidade de parâmetros
- Se possui retorno ou não
- Se é recursiva ou não

Quando for especificado que a solução deve ter uma função recursiva, é levando em consideração que a recursão é direta ou simples, ou seja, que a função seja diretamente invocada dentro dela mesma.

Como uma tentativa de garantir que a função seja usada, este *tree walker* verifica se a função foi invocada em alguma parte do código-fonte que não seja dentro dela mesmo, exceto nos casos de função recursiva que deve ser invocada dentro e fora dela.

As listagens 5.11 e 5.12 apresentam duas soluções com uma função recursiva para calcular o fatorial de um número. Para situações como essas é interessante que o professor

também limite o número de laços de repetição para 0 com o objetivo de que ele realmente não faça uso de *for* ou *while*.

```
def fatorial(n):
    if n <= 1:
        return 1
    else:
        return n * fatorial(n - 1)
print(fatorial(5))
```

Listagem 5.11 – Solução com uma função recursiva em Python

```
package codigo.exemplo.dissertacao;
public class ExemploFuncao {
    public static int fatorial(int num) {
        if (num <= 1)
            return 1;
        else
            return fatorial(num - 1) * num;
    }
    public static void main(String[] args) {
        System.out.println(fatorial(5));
    }
}
```

Listagem 5.12 – Solução com uma função recursiva em Java

### 5.3.4 Blocos Vazios

Este Tree Walker verifica se o código-fonte possui blocos vazios. Em Python, blocos vazios podem ser construídos por meio do comando *pass*. A listagem 5.13 apresenta um código-fonte com um bloco vazio em Python.

```
x = int(raw_input())
y = int(raw_input())
def somar(x, y):
    pass
print (x+y)
```

Listagem 5.13 – Solução com bloco vazio em Python

Em Java, um bloco vazio é identificado quando não existe comando entre os *tokens* { e }. A listagem 5.14 apresenta um exemplo de bloco vazio na função *somar()*, entretanto poderia ser um bloco *if*, *else*, *for*, *while* ou outros.

```
package codigo.exemplo.dissertacao;
public class ExemploBlocoVazio {
```

```
3     public void somar(int num1, int num2) {
4         // Bloco vazio
5     }
6 }
```

Listagem 5.14 – Solução com bloco vazio em Java

### 5.3.5 Array/Matriz

De acordo com o levantamento feito no capítulo 4, a necessidade de especificação de restrições relacionadas a Array/Matriz é maior do que as outras. Diante disso, este *Tree Walker* permite que o professor expresse se a solução deve ser obrigada ou deve ser proibida de utilizar Array/Matriz.

A utilização de vetores em Java é bastante comum e possui várias formas de declaração como apresenta a listagem 5.15.

```
1 package codigo.exemplo.dissertacao;
2 public class ExemploVetor {
3     public static void main(String args[]) {
4         int[] declaracao1_vetor;
5         int[][] declaracao1_matriz;
6 
7         int declaracao2_vetor[];
8         int declaracao2_matriz[][];
9 
10        int[] declaracao3_vetor = {1,2,3};
11        int declaracao3_matriz[][] = {{1,2,3},{0,1,3},{0,0,-1}};
12    }
13 }
```

Listagem 5.15 – Diversas formas de declarar vetor em Java

O analisador estático também verifica se há declaração de vetores nos parâmetros dos métodos, exceto o *main(String args[])* que é padrão de toda aplicação Java.

No universo das linguagens de programação, nem todas possuem o conceito de vetores tão explícito da forma como Java, C, C++ e C# possuem. Um exemplo disso é a linguagem Python, nela quando é desejado resolver um problema que necessita de uma estrutura sequencial, geralmente as listas são utilizadas. Dessa forma, a aplicação deste *Tree Walker* em soluções Python buscará por declarações de listas como apresenta a listagem 5.16.

```
1 lista_1 = list() # Declaracao explicita de uma lista
2 lista_2 = [] # Declaracao explicita de uma lista
3 lista_3 = range(1,5) #Retorna a lista [1,2,3,4]
4 notas = "10 9.5 8"
```

```
5 lista_4 = notas.split(" ") #Retorna a lista [10, 9,5, 8]
```

Listagem 5.16 – Declaração de listas em Python

### 5.3.6 Listas e Dicionários

Esta abordagem também permite ao professor especificar que a solução do aluno precisa ter ou não deve ter uma lista ou um dicionário. A listagem 5.17 possui três declarações de listas e três declarações de dicionários que o analisador estático consegue identificar na linguagem Python.

```
1 #Tres declaracoes explicitas de listas
2 lista_1 = list()
3 lista_2 = []
4 estados = ['Sergipe','Alagoas','Pernambuco']
5 #Tres declaracoes explicitas de dicionarios
6 dicionario = dict()
7 dicionario_2 = {}
8 ing2esp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Listagem 5.17 – Solução utilizando uma lista

O analisador estático considera a declaração de três classes como listas em Java: *ArrayList*, *LinkedList* e *Vector*. Em relação ao dicionário são consideradas estas classes: *HashMap*, *TreeMap* e *Hashtable*. O processo para adicionar novas classes é trivial e pode ser utilizado futuramente. A listagem 5.18 apresenta a utilização de todas essas classes que podem ser identificadas pelo analisador estático.

```
1 package codigo.exemplo.dissertacao;
2 import java.util.ArrayList;
3 import java.util.HashMap;
4 import java.util.Hashtable;
5 import java.util.LinkedList;
6 import java.util.TreeMap;
7 import java.util.Vector;
8 public class ExemploListaDicionario {
9  public static void main(String args[]) {
10   // Declaracao de dicionarios
11   Hashtable<String, String> table = new Hashtable<String, String>();
12   HashMap<String, String> map = new HashMap<String, String>();
13   TreeMap<String, String> tree = new TreeMap<String, String>();
14
15   // Declaracao de listas
16   LinkedList<Integer> lista_inteiros = new LinkedList<Integer>();
17   ArrayList<String> lista_nomes = new ArrayList<String>();
```

```
18   Vector<Double> lista_double = new Vector<Double>();
19  }
20 }
```

Listagem 5.18 – Diversas formas de declarar listas e dicionários em Java

Neste ponto, é importante destacar que a decisão de apenas verificar a declaração e não o uso de lista/dicionário está baseada em observações feitas durante a análise das submissões dos alunos. Foi percebido que em muitas ocasiões os alunos declaravam mais de uma lista/dicionário, mas não utilizavam todos eles. Diante deste fato, foi decidido não prejudicar a correção da submissão do aluno se ele conseguiu declarar corretamente alguma lista/dicionário.

### 5.3.7 Chamada de função

Este *Tree Walker* tem como objetivo verificar se o código-fonte está invocando alguma função. Esse procedimento é interessante porque em algumas situações o professor pode querer que o aluno busque o valor máximo de uma lista sem utilizar a função *max()*, calcule a raiz quadrada sem utilizar a função *sqrt()*.

Por outro lado, o professor pode também exigir que o aluno invoque essas funções para resolver os problemas, isso é comum para funções que envolvem manipulação de *string*, como *replace()*, *replaceFirst()*, *subSequence()*, *split()*, *substring()*, *toLowerCase()*, *toUpperCase()*, *trim()* e outras. Portanto, este *Tree Walker* permite que o professor especifique que a invocação de uma determinada função é obrigatória ou proibida.

### 5.3.8 Integração com o The Huxley

A abordagem apresentada neste trabalho foi integrada ao sistema de juiz *on-line* The Huxley, o que demandou sucessivas reuniões com sua equipe de desenvolvimento. A figura 20 ilustra o novo processo de avaliação de soluções do The Huxley. O programa somente será avaliado estaticamente se retornar as respostas corretas para os testes caixa-preta que já são implementados tradicionalmente em todos os sistemas de juiz *on-line*

A equipe de desenvolvimento do The Huxley implementou as interfaces mostradas nas figuras 21 e 22 e adaptou o sistema para que funcionasse com essa nova abordagem de correção de problemas. A disponibilização de uma interface para especificação de restrições é um dos objetivos deste trabalho. Após o preenchimento da interface, o The Huxley produz um arquivo *Json*, de acordo com o formato descrito no apêndice B, para que, em conjunto com a solução do aluno, sejam passados como as entradas do analisador estático.

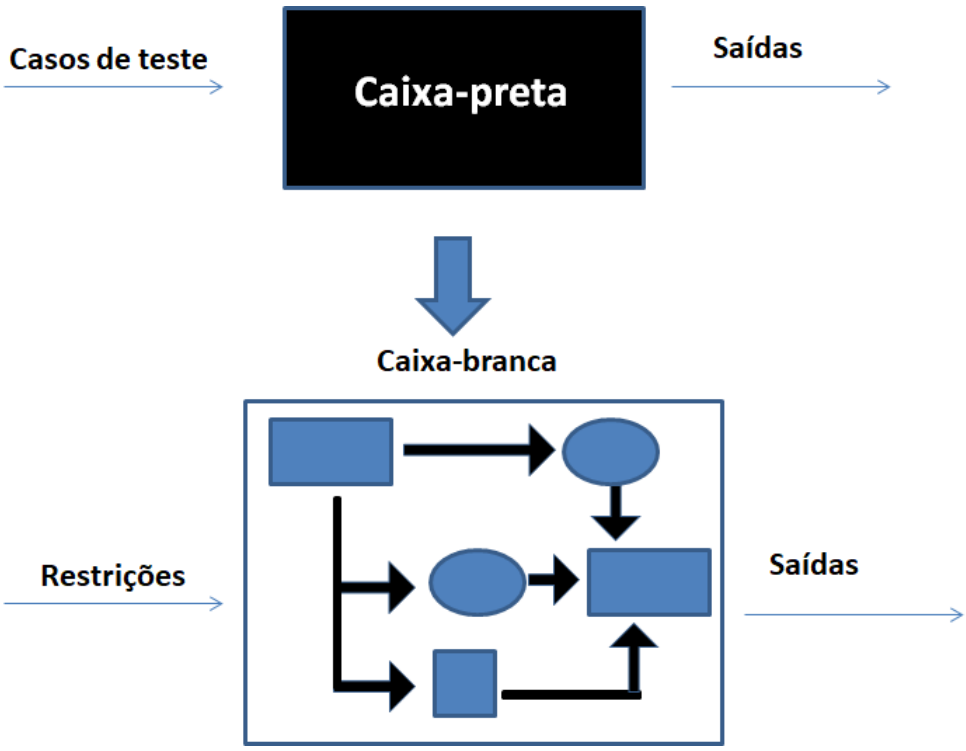


Figura 20 – Novo processo de avaliação do The Huxley


Figura 21 – Interface para especificação de restrições

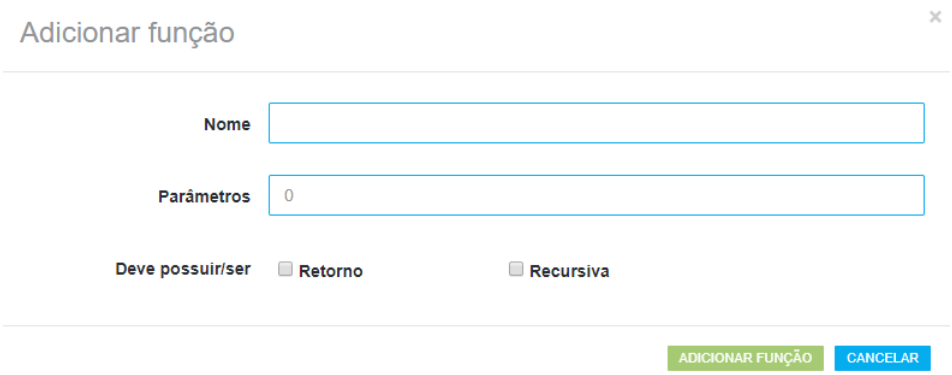


Figura 22 – Interface para especificação de restrição

# 6

# Avaliação da Abordagem Proposta

Este capítulo apresenta duas formas para avaliação da proposta desenvolvida. A primeira delas está na seção 6.1 e consiste em uma apresentação e posterior comparação dos trabalhos relacionados encontrados na literatura com a proposta desenvolvida. A segunda está na seção 6.2 e consite em uma aplicação de um *survey* com a finalidade de descobrir o grau de interesse de professores que lecionam ou lecionaram introdução à programação em utilizar uma ferramenta com funcionalidades semelhantes à que foi desenvolvida nesta pesquisa.

## 6.1 Trabalhos Relacionados

Este trabalho contou com o estudo de duas revisões sistemáticas. Uma sobre técnicas de análise estática (RAHMAN; NORDIN, 2007) e outra sobre juízes *on-line* no ensino de programação introdutória (FRANCISCO; JúNIOR; AMBROSIO, 2016). Apesar de uma das revisões ser relativamente recente (2016), ainda foram realizadas buscas nas bases dos periódicos IEEE e ACM com o objetivo de encontrar trabalhos que utilizassem técnicas de análise estática para avaliar soluções para problemas de programação.

Esta seção apresenta três abordagens não contempladas nas revisões sistemáticas (RAHMAN; NORDIN, 2007) e (FRANCISCO; JúNIOR; AMBROSIO, 2016), mas que estão alinhadas com o perfil deste trabalho. Os critério para estudar estas abordagens foram: utilizar alguma técnica de análise estática e ser utilizada em turmas introdutórias de programação. Na seção 6.1.4.3 deste capítulo é realizada uma comparação das três abordagens com a desenvolvida nesta pesquisa.

### 6.1.1 Scheme-robo

Scheme-robo (SAIKKONEN; MALMI; KORHONEN, 2001) é um sistema de avaliação automática de exercícios de programação para a linguagem Scheme utilizado completamente

sem interferência humana. O sistema possui um conjunto de funcionalidades que engloba as análises dinâmica e estática de avaliação de código.

O sistema funciona *on-line* por meio do envio de *e-mail*. Para saber se uma solução está correta, o aluno deve enviar um *e-mail* com o seu código e aguardar uma resposta após 10 minutos. É importante ressaltar que o sistema não leva esse tempo para analisar a solução. Na verdade esse tempo foi escolhido para que o aluno não enviasse várias submissões sucessivas sem pensar nas alterações que está realizando no código-fonte. Nessa mesma linha de raciocínio, o sistema limita o número de submissões em 20 vezes por problema.

Para adicionar um novo problema no Scheme-robo, o professor precisa escrever um arquivo de configuração com expressões Lisp que descreve como a solução deve ser analisada. Esse arquivo geralmente possui entre 20 e 100 linhas e contém os casos de teste, modelos de soluções, palavras-chaves proibidas ou necessárias e outras informações.

Uma das formas de analisar uma solução por meio do Scheme-robo é verificar se alguns comandos estão presentes na solução do aluno. Depois de realizada essa busca, o professor pode decidir por validar ou invalidar a solução de acordo com seu propósito. A listagem 6.1 apresenta uma padrão para especificação de restrição que permite checar se determinado comando está presente na solução.

```
1 if(keyword <symbol> ...)
2 (<points> [<comment> ...])
3 (<points> [<comment> ...])
```

Listagem 6.1 – Especificação de palavra-chave

Onde <*symbol*> é qualquer simbolo Lisp, <*points*> pode ser um inteiro ou um símbolo e <*comment*> representa uma *string* opcional utilizada para exibir algum *feedback* para o aluno. A linha 2 é executada se a palavra-chave for encontrada e a linha 3 caso contrário.

Um exemplo de utilização da estrutura acima é apresentado na listagem 6.2

```
1 (if (keyword set!))
2 (fail ''Don't use set! in this exercise!'')
3 ((0))
```

Listagem 6.2 – Exemplo de especificação

A listagem 6.2 especifica que deve ser procurado o símbolo set! na solução submetida e envia um *feedback* para o aluno se o símbolo for encontrado invalidando a resposta. É importante ressaltar que a palavra-chave set! corresponde ao símbolo set! no código-fonte do aluno e não a um trecho de *string* que contém a palavra set! ou um símbolo longo como foo-set!.

Esta abordagem fornece um grau de granularidade refinado ao professor. Por meio dela o professor pode checar diversos aspectos do código do aluno, entretanto para isso ele deve utilizar muitas linhas de código-fonte, transformando-se em uma solução pouco prática. Além disso, é

notável que esta abordagem possui pouca utilidade prática porque a linguagem Scheme é pouco utilizada em cursos de introdução à programação.

### 6.1.2 Projekt Tomo

Projekt Tomo (JERSE; LOKAR, 2016) é um serviço web para correção automática de problemas de programação. O sistema é utilizado por estudantes dos cursos de Matemática e Física da Universidade de Ljubljana, na Eslovênia.

Após os alunos se autenticarem no sistema, é disponibilizada uma tela com um conjunto de problemas em seus respectivos cursos (alguns problemas podem ser escondidos pelos professores). Para o aluno resolver um problema ele deve baixar um arquivo que contém a descrição do problema e todos os casos de testes necessários para a validação. Neste sistema as submissões dos alunos são testadas localmente, sem a necessidade de carregar ou copiar programas para um ambiente de codificação diferente.

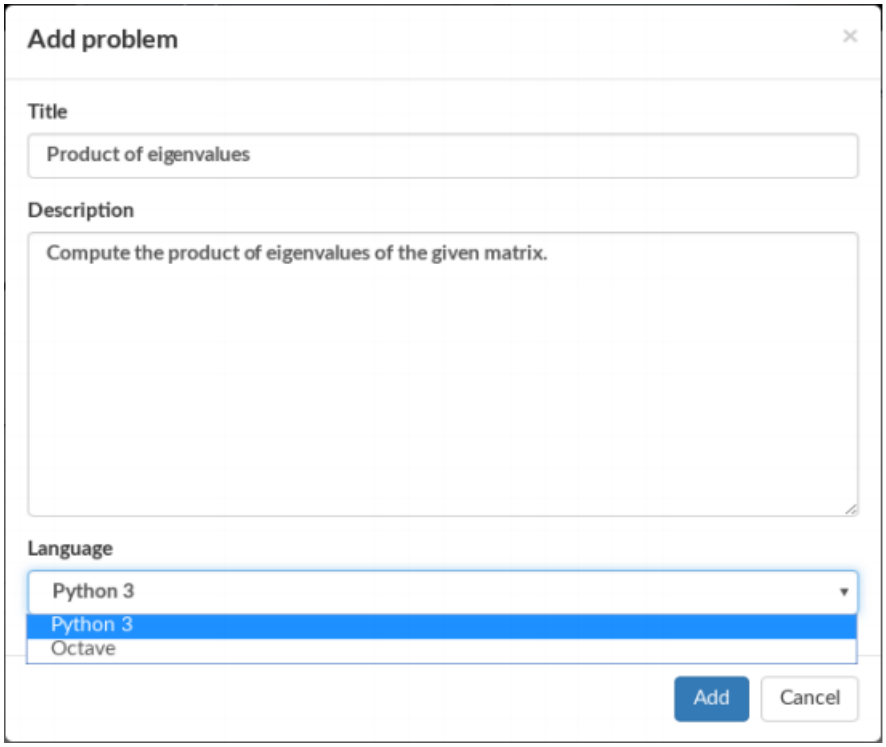


Figura 23 – Tela para adicionar um problema

Para o professor adicionar um problema ele precisa preencher um formulário como o da figura 23 com o título, a descrição do problema e a linguagem de programação. Após esse processo, será gerado um arquivo que o professor deverá baixar para adicionar novas informações ao problema. Com esse arquivo é possível adicionar os casos de testes e complementar com código-fonte para realizar análises estáticas sobre o código-fonte do aluno. Após esse processo, o professor submete o arquivo que representa o problema para que possa ser baixado pelos alunos.

Dois exemplos de análises que podem ser feitas pelo professor são apresentadas nas listagens 6.3 e 6.4. Para que os testes estáticos funcionem o professor precisa adicionar esses trechos de código no arquivo do problema.

```
1 sol = check.current_part.solution;
2 if strfind(sol, ``det'')
3   check_error(``Your solution should not contain the string 'det.'");
```

Listagem 6.3 – Análise estática para solução Octave

A listagem 6.3 está verificando se o aluno utilizou a *string* det na solução e enviando uma mensagem de erro caso encontre.

```
1 tree = ast.parse(Check.current_part['solution'])
2 for node in ast.walk(tree):
3   if isinstance(node, ast.Call):
4     name = node.func.id
5     if name == 'det':
6       Check.error('det function should not be used')
```

Listagem 6.4 – Análise estática para solução Python

A listagem 6.4 está fazendo uma análise mais rebuscada do que simplemente uma busca por *string*. Nesse caso o professor está interessado em descobrir se o aluno está chamando alguma função com o nome de "det". Para isso é utilizada a biblioteca **AST** (PYTHON, 2011) da própria linguagem de programação Python.

Este caso é interessante porque como o professor dispõe de uma biblioteca que gera a árvore sintática abstrata do código-fonte, então é oferecido a ele um grau de granularidade refinado. Essa granularidade permite que ele verifique quaisquer construções existentes na AST, por exemplo: quantidade de *ifs*, *for*, *while*, definição de função, chamada de função, definição de classe, declaração de variável, escopo de variável e outros. Para isso, no entanto, ele precisa conhecer a fundo o funcionamento dessa API e criar um programa de avaliação estática para cada exercício.

Esta abordagem certamente demanda muito trabalho de codificação e exige conhecimento profundo da AST da linguagem de programação. Além disso, a checagem deve ser reescrita para cada linguagem de programação, o que por si só já é um desafio.

### 6.1.3 Portugol Studio

O Portugol Studio é um ambiente para aprender a programar, voltado para os iniciantes em programação que falam o idioma português. Possui uma sintaxe fácil, diversos exemplos e materiais de apoio à aprendizagem. Também possibilita a criação de jogos e outras aplicações. Atualmente o sistema conta com 72090 usuários cadastrados (STUDIO, 2017). O sistema é desenvolvido pela Universidade do Vale do Itajaí - SC, possui código-fonte aberto no GitHub

e trabalhos acadêmicos (PELZ; RAABE, 2013) (PELZ; JESUS; RAABE, 2012) já foram realizados nele.

Conforme apresentado em (PELZ, 2014), foram realizadas diversas alterações no sistema com o objetivo de aprimorar o processo de correção automática de problemas. Uma das alterações efetuadas procurava identificar construções obrigatórias ou proibidas no código-fonte. Para isso foi necessária a implementação de *Tree Walkers*, que são algoritmos que percorrem a Árvore Sintática Abstrata da solução. Esse processo foi realizado com auxílio do framework ANTLR 3 (PARR, 2007).

O trabalho ainda conta com a capacidade de comparação da solução do aluno com soluções modelo enviadas pelo professor. Para fazer a comparação, as soluções enviadas pelo professor e a do aluno são normalizadas, então é aplicado o algoritmo da distância Levenshtein para descobrir o grau de semelhança entre elas. O quadro 6.1 apresenta os *Tree Walkers* implementados em (PELZ, 2014).

| **Tree Walkers** | **Parâmetros** | **Objetivo** |
|---|---|---|
| MandatoryInstructions | Lista de nós que devem aparecer na estrutura do programa. | Procura a existência de uma ou mais estruturas dentro do código do aluno. Realiza uma notificação quando não encontrado. |
| ProhibitedInstructions | Lista de nós que não podem aparecer na estrutura do programa. | Realiza uma notificação quando encontra um ou mais instruções na estrutura do aluno. |
| ReadAfterOperation | Não possui | Realiza uma notificação caso o aluno faça uma entrada de dados após ter manipulado as variáveis do programa. |
| ReadWriteOrder | Não possui | Realiza uma notificação ao perceber que a ordem das variáveis passadas por parâmetro no comando leia está em uma ordem diferente durante a saída de dados. |
| UsingAux | Não possui | Realiza uma varredura na estrutura do código para certificar-se que o uso de variável auxiliar para a troca de valores entre variáveis foi utilizada corretamente. |
| VarIsNotUsed | Não possui | Realiza a notificação de alguma variável que foi declara e não utilizada durante o programa. |

Quadro 6.1 – Tree Walkers desenvolvidos em (PELZ, 2014)

Uma evolução deste trabalho foi realizada por (HODECKER, 2014) que criou um editor de questões e implementou outros *Tree Walkers* para facilitar a criação de problemas por outros professores. O quadro 6.2 apresenta os *Tree Walkers* implementados em (HODECKER, 2014).

| Tree Walkers | Parâmetros | Objetivo |
|---|---|---|
| PossibleInfiniteLoop | Não possui | Identificar possíveis laços infinitos. |
| NumberOfConditions | Inteiro: Número correto de possibilidades esperadas. | Verificar se existe uma quantidade errada de desvios no código do aluno |
| ConstantIndex | Não possui | Detectar se o aluno não incrementou o índice do vetor ou matriz. |
| EmptyBlocks | Não possui | Detectar se existe algum bloco de código vazio no código do aluno. |
| NumberOfInstructions | Inteiro: Número máximo permitido; Bloco da ASA: Instrução analisada. | Detectar se o aluno está utilizando uma instrução mais vezes que o necessário. |
| MandatoryArray | Inteiro: 1 para vetor e 2 para matriz. | Detectar se existe vetor ou matriz no exercício. |

Quadro 6.2 – Tree Walkers desenvolvidos em (HODECKER, 2014)


Figura 24 – Cadastro de casos de teste

A figura 24 apresenta a interface gráfica utilizada pelos professores para adicionar os casos de testes de cada problema. Neste exemplo foi adicionado um caso de teste com dois valores de entrada dos tipos inteiro e real e apenas um valor de saída do tipo inteiro. Esse recurso é utilizado pelo Portugol Studio para realizar uma análise dinâmica sobre a solução do aluno.

A figura 25 apresenta a interface gráfica utilizada pelos professores para adicionar

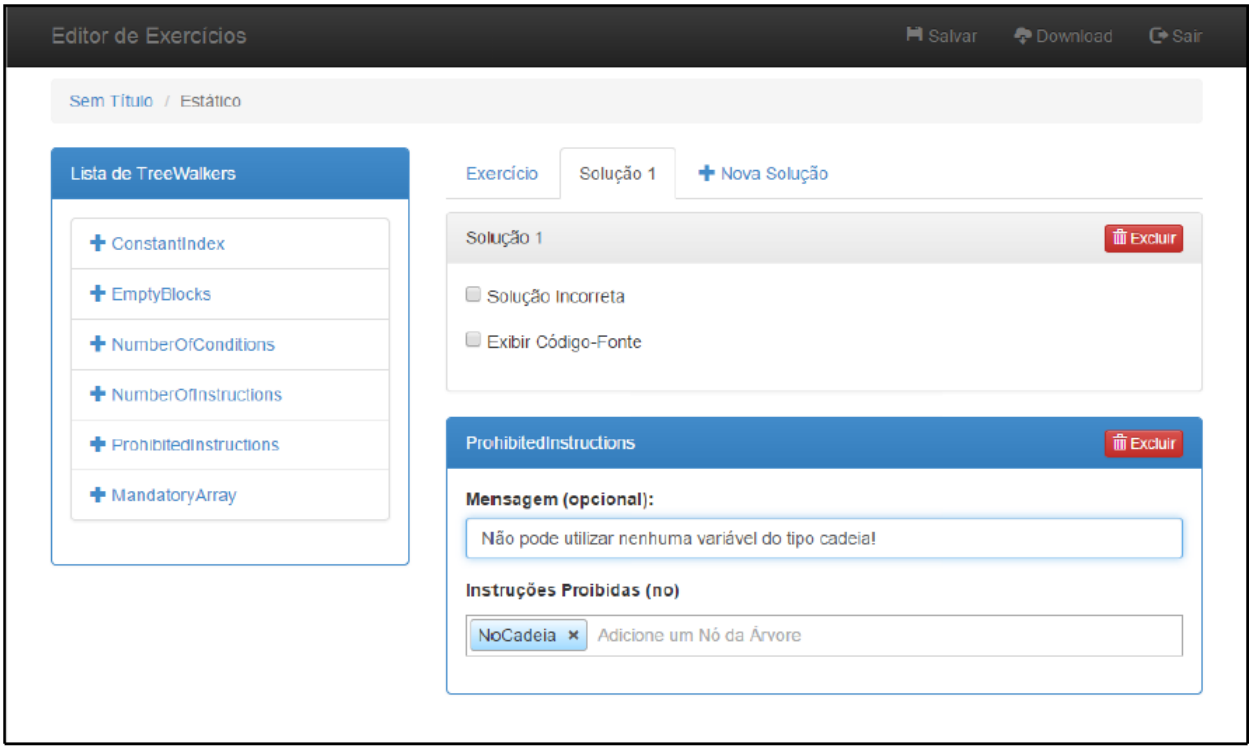


Figura 25 – Interface gráfica para manipulação dos Tree Walkers

restrições nos problemas. Neste exemplo está sendo especificada uma restrição para que o código-fonte do aluno não possua variáveis do tipo cadeia. Para verificar se a solução está de acordo com a restrição foi utilizado o *Tree Walker ProhibitedInstructions* que percorre a Árvore Sintática do código-fonte em busca do nó "NoCadeia". Se ele for encontrado, a mensagem "Não pode utilizar nenhuma variável do tipo cadeia" é apresentada para o aluno.

### 6.1.4 Comparação

O quadro 6.3 apresenta alguns critérios de uma comparação entre as abordagens apresentadas neste capítulo com a abordagem desenvolvida neste trabalho.

| Plataforma | Linguagem | Facilidade de uso | Grau de flexibilidade | Abordagem unificada |
|---|---|---|---|---|
| **Scheme-robo** | Scheme | baixa | elevado | não |
| **Projekt Tomo** | Octave/ Python | baixa | elevado | não |
| **Portugol Studio** | Portugol | alta | baixo | não |
| **The Huxley** | Java/ Python | alta | adequado | sim |

Quadro 6.3 – Comparação entre as abordagens

#### 6.1.4.1 Facilidade de uso

A facilidade de uso é um dos principais critérios a ser analisado, pois não adianta uma abordagem oferecer inúmeros recursos se os mesmos não forem fáceis de utilizar. As plataformas

Scheme-robo e Projekt Tomo estão classificadas com uma baixa facilidade de uso porque em ambas o professor precisa dispor de recursos de programação para que a avaliação estática seja realizada. Esta exigência não é trivial porque a tarefa de avaliação estática requer um compromisso do professor de imaginar e cercar por meio de uma solução programática os diversos cenários que a solução do aluno pode cumprir com a especificação do professor, mas ainda assim não estar de acordo com o que o mesmo pretendia. Alinhado a esse esforço está a necessidade de criar um ou mais programas (depende da quantidade de linguagens) para cada exercício. Por exemplo, para criar um questionário com 10 problemas com a abordagem Scheme-robo o professor precisa de 10 programas que realizem a avaliação estática, o caso do Projekt Tomo é ainda mais delicado porque se o professor disponibilizar o problema para duas linguagens, o mesmo deverá criar 20 programas diferentes.

A plataforma Portugol Studio, assim como a abordagem desenvolvida neste trabalho, não possui o problema da dificuldade de uso apresentado nas outras plataformas. Nessa abordagem, foi desenvolvida a ideia de criação de *Tree Walkers* com comportamentos padronizados que apenas necessitam de parâmetros fornecidos pelo professor para que a avaliação estática seja realizada.

#### 6.1.4.2 Grau de flexibilidade

O grau de flexibilidade pode ser definido como o conjunto de restrições que é disponibilizado para que o professor utilize. Quanto maior for o conjunto de **restrições significativas**, melhor pode ser considerada a plataforma neste critério. O termo restrições significativa está enfatizado porque simplesmente oferecer um grande conjunto de restrições não garante que ele seja frequentemente utilizado. Na verdade, esse conjunto deve ser composto por restrições constantemente demandadas por professores no ensino de Introdução à Programação.

As abordagens Scheme-robo e Projekt Tomo possuem um elevado grau de flexibilidade porque oferecem ao professor a opção de programar como a solução deve ser avaliada.

O Portugol Studio possui um grau de flexibilidade baixo porque não oferece um grande conjunto de restrições significativas de acordo com o levantamento realizado no capítulo 4 e com o *survey* apresentado no capítulo 6.1.4.3. Essa abordagem, por exemplo, não permite ao professor especificar que um problema deve ser resolvido com uma função com um determinado nome, quantidade de parâmetros, com ou sem retorno e recursiva. Entretanto, como a abordagem permite proibir algumas construções, é fácil perceber que se for especificado que a solução não pode utilizar laços de repetição, uma saída elegante para o problema é utilizar a recursão. Esse raciocínio pode resolver o problema em parte porque em um mesmo exercício o professor pode desejar não só que seja utilizada uma função recursiva, mas também um laço de repetição. Esse caso não é suportado pela abordagem Portugol Studio.

A abordagem desenvolvida neste trabalho possui um grau de flexibilidade **adequado** porque está alinhada com a maior parte das restrições levantadas no capítulo 4 e com o resultado

do *survey* apresentado na seção 6.1.4.3. Apenas duas restrições catalogadas no capítulo 4 não são suportadas pela nossa proposta: *do while* e *switch*. Essas restrições não foram adicionadas porque não estão presentes em todas as linguagens de programação incluídas no escopo deste trabalho. Além disso, elas representam pouco (apenas duas ocorrências) no universo de todas as restrições catalogadas.

#### 6.1.4.3 Abordagem unificada

Este critério diz respeito à forma como o modelo de avaliação estática foi projetado em cada abordagem. Uma abordagem é considerada unificada se o seu modelo foi concebido para trabalhar eficientemente com várias linguagens de programação. Algumas abordagens apresentadas neste capítulo permitem que o aluno submeta a solução em apenas uma linguagem de programação. Esta particularidade pode ter interferido no momento de definir como seria projetado o modelo de avaliação estática. Entretanto, no contexto dos juízes *on-line*, geralmente mais de uma linguagem de programação são disponibilizadas para o aluno e uma abordagem unificada apresenta-se como mais adequada.

A abordagem Scheme-robo não é considerada unificada porque seu modelo de avaliação estática é personalizado para a linguagem funcional Scheme. Para o professor proibir ou exigir algum tipo de construção em uma solução Scheme, o mesmo deve criar um arquivo de configuração explicitando quais comandos ele deseja que sejam procurados na AST do código-fonte. Essa característica torna a abordagem personalizada para um tipo de linguagem.

A abordagem Projekt Tomo não é considerada unificada porque possui mais de um modelo de avaliação estática por linguagem de programação. Nessa abordagem, são oferecidos ao professor todos os recursos da própria linguagem em questão. Essa característica permite que análises diferentes sejam feitas em diferentes soluções de acordo com a linguagem de programação. Isso fere o conceito de abordagem unificada porque uma linguagem pode oferecer mais recursos do que outra e demanda codificação específica.

O Portugol Studio apresenta um modelo de especificação de restrições de alto nível que abstrai do professor os detalhes de programação. Entretanto, para que o mesmo especifique que uma determinada construção deve ou não deve ser utilizada, o mesmo precisa conhecer os detalhes da AST de uma solução Portugol. Essa característica limita a especificação a um de tipo linguagem.

A abordagem proposta por este trabalho foi projetada com o objetivo de suportar várias linguagens de programação e não apenas para Python e Java. Por exemplo, se um professor sabe como especificar restrições para a linguagem Java hoje e futuramente a abordagem evoluir e passar aceitar novas linguagens orientadas a objeto como C++ e C#, nada mudará na maneira de especificar as restrições. Isso ocorre porque antes da abordagem ser desenvolvida foram consideradas quais características/restrições poderiam ser mapeadas para outras linguagens de programação. Diante dessa escolha, a API desenvolvida foi projetada utilizando os conceitos do

padrão de projeto *Abstract Factory*, conforme apresentado no capítulo 5.

## 6.2 Validação da proposta

Este seção apresenta e discute as respostas de um questionário que foi aplicado para 38 professores com experiência em turmas introdutórias de programação. Houve uma divulgação na lista de discussão da Sociedade Brasileira de Computação, o que permitiu que qualquer professor interessado participasse. O questionário foi feito por meio da plataforma Google Forms e disponibilizado por *e-mail* para profissionais da área. O principal objetivo foi verificar se o que fora desenvolvido neste trabalho é relevante e supre as principais necessidades dos professores. Além disso, visava descobrir se há outros fatores que não foram considerados nesta pesquisa.

### 6.2.1 Questionário aplicado

O questionário foi dividido em 3 seções, as quais serão apresentadas nas subseções a seguir. Cada seção do questionário contém explicações e perguntas. As respostas dos professores e uma discussão acerca das mesmas são apresentadas à medida que as perguntas são listadas.

**A maioria das perguntas possui 5 alternativas numeradas de 1 a 5. Nestes casos, 1 significa que o professor não usaria a funcionalidade e 5 significa que o professor usaria com frequência a funcionalidade.** Neste questionátio, há 6 perguntas relacionadas a lista, dicionário e tupla que estão classificadas como opcionais porque esses conceitos são implementados através de bibliotecas nas linguagens Java, C, C++ e C#, ao contrário de Python.

#### 6.2.1.1 Critérios para analisar automaticamente uma solução para um problema de programação

A figura 26 apresenta a tela inicial do questionário. Todos os professores concordaram com os termos apresentados.

#### 6.2.1.2 Pesquisa - Critérios

A figura 27 apresenta a principal pergunta que deve ser levada em consideração para responder todos os itens do questionário.

#### 6.2.1.3 Perguntas e respostas obtidas

Com qual frequência você usaria cada funcionalidade oferecida por essa ferramenta?

**Q1. Especificar que o aluno deve criar e utilizar uma função com um determinado nome e uma determinada quantidade de parâmetros?**

**Q2. Especificar que o aluno deve criar e utilizar uma função com/sem retorno?**

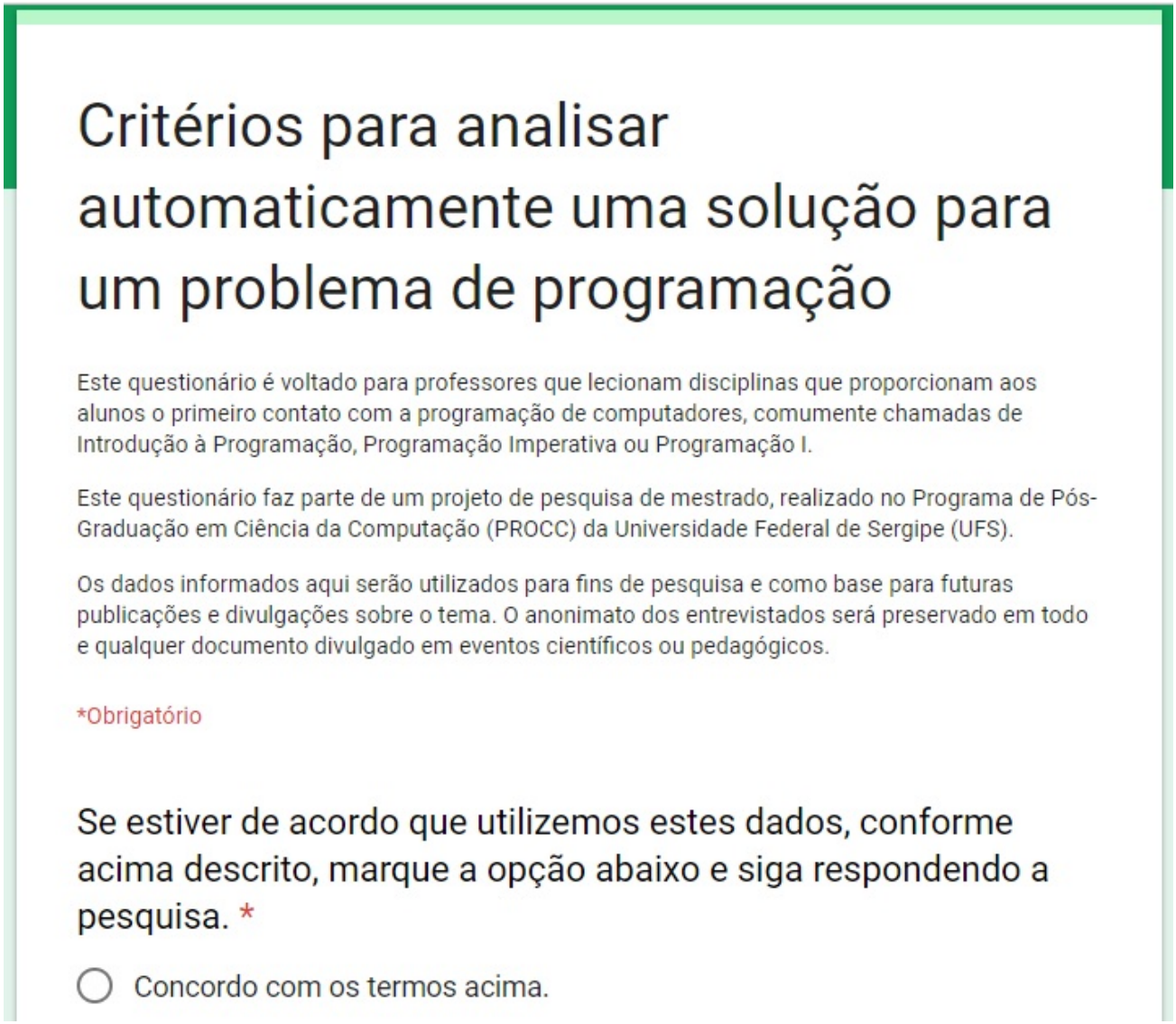


Figura 26 – Tela do questionário

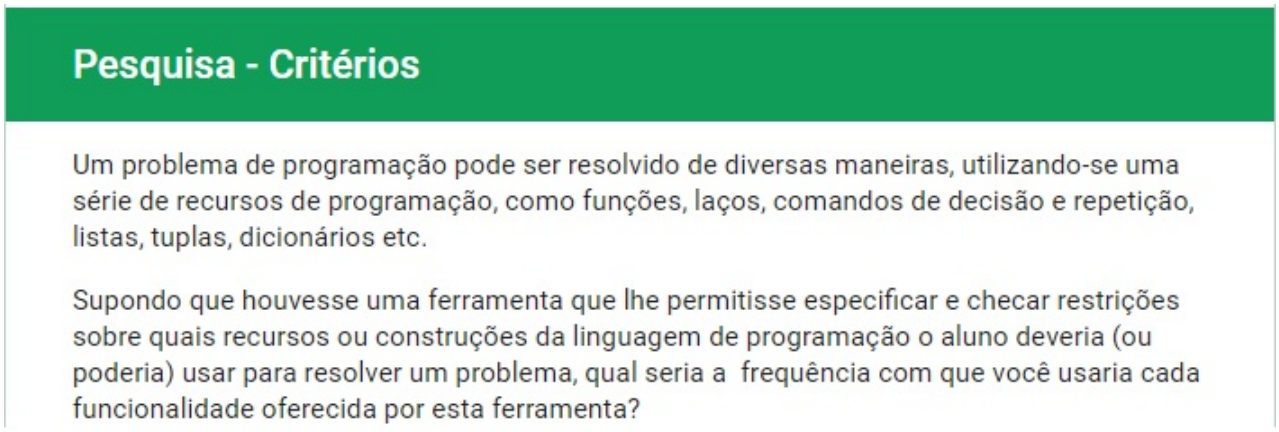


Figura 27 – Tela do questionário

**Q3. Especificar que o aluno deve criar e utilizar uma função recursiva?**

As três primeiras perguntas do questionário estão relacionadas à criação e utilização de

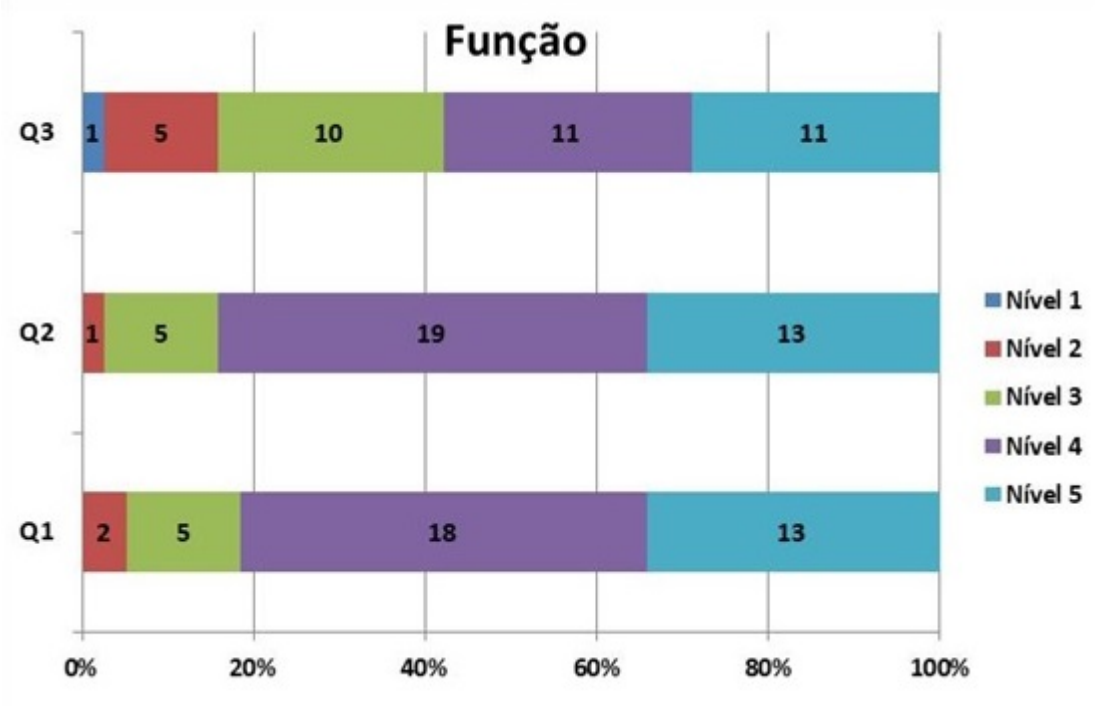


Figura 28 – Resultado relacionado à criação e utilização de função

funções. Suas respectivas respostas são apresentadas na figura 28. As respostas para a pergunta **Q1** estão notavelmente agrupadas nos níveis 4 e 5. Isso significa que a maioria dos professores que responderam ao questionário entende que realmente há uma grande necessidade de forçar o aluno a modularizar sua solução. Este resultado alinhado com o mapeamento de funções realizado no capítulo 4 demonstram que a avaliação de funções por uma abordagem estática é um fator de fundamental importância.

Na linguagem Java, a assinatura de um método é composta pelo seu nome, tipos de parâmetros, quantidade de parâmetros e ordem dos parâmetros. O tipo de retorno do método não é avaliado para diferenciar um método de outro, portanto não é possível ter dois métodos com os mesmos nomes, quantidades, tipos e ordem de parâmetros, ainda que eles tenham tipos de retornos diferentes (em uma mesma classe). Esta foi a motivação para criar a questão **Q2** abordando apenas o retorno do método.

Apesar disto, o resultado não foi muito distante do apresentado na questão anterior. Na verdade, os dois resultados podem ser explicados pela necessidade que os professores têm de forçar o aluno a modularizar suas soluções. Observando-se a figura 28, 50% das respostas estão no nível 4, apenas uma resposta no nível 2 (2.6%), 5 respostas no nível 3 (13.2%) e 13 respostas no nível máximo (34.2%).

Em relação a questão **Q3**, o resultado não foi tão claro quanto os das duas anteriores, mas ainda assim é possível notar uma aproximação aos níveis mais elevados de possível utilização da funcionalidade. Este resultado pode ser explicado pela necessidade que o professor tem de fazer o aluno entender que apesar de uma solução iterativa ser mais fácil

de implementar, é preciso também desenvolver o raciocínio da recursão.

Considerando a figura 28, apenas 1 (2,6%) professor apontou que não usaria a funcionalidade, 5 (13,2%) classificaram como nível 2, 10 (26,3%) como nível 3, 11 (28,9%) como nível 4 e novamente 11 (28,9%) como nível 5.

**Q4. Especificar que o aluno DEVE UTILIZAR uma determinada função da API/Biblioteca da linguagem para resolver o problema?**

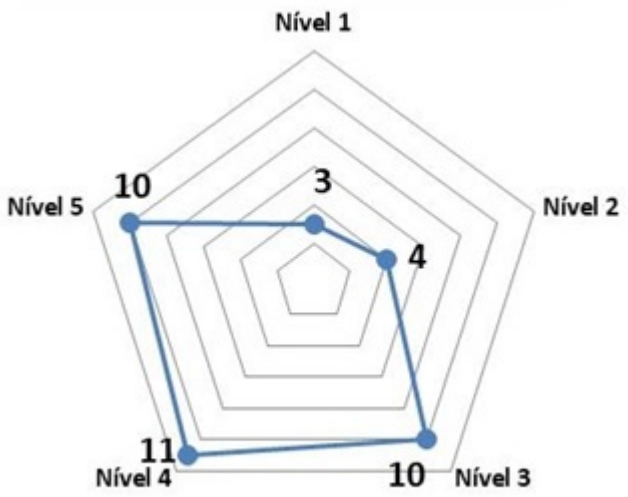


Figura 29 – Resultado relacionado à utilização de uma API

O resultado desta questão aponta que forçar o aluno a responder um problema com uma determinada função é uma funcionalidade interessante e desejada por muitos professores. Conforme a figura 29, apenas 7 (18,4%) professores responderam com uma classificação baixa (3 nível 1 e 4 nível 2) e 31 (81,6%) responderam com uma classificação maior ou igual a 3 (10 nível 3, 11 nível 4 e 10 nível 5).

**Q5. Especificar que o aluno NÃO DEVE UTILIZAR uma determinada função da API/-Biblioteca da linguagem para resolver o problema?**

Se por um lado a ideia de exigir que o aluno utilize uma função da API foi apreciada por uma grande parte dos professores, a necessidade da proibição de utilizá-la foi ainda mais apreciada. Isto ocorre porque atualmente as prinicipais linguagens de programação possuem um grande conjunto de funções implementadas que resolve grande parte dos problemas introdutórios de programação. Algumas delas são: *max()*, *min()*, *sum()*, *sort()*, *sqrt()* e *size()*. Diante desse cenário, proibir o aluno de utilizar alguma função foi encarado como uma funcionalidade interessante. Observando-se a figura 30, apenas 2 (5,3%) professores disseram que não usariam a funcionalidade, outros 2 (5,3%) classificaram com o nível 2 e 5 (13,2%) responderam com o nível intermediário. Novamente, a maioria se

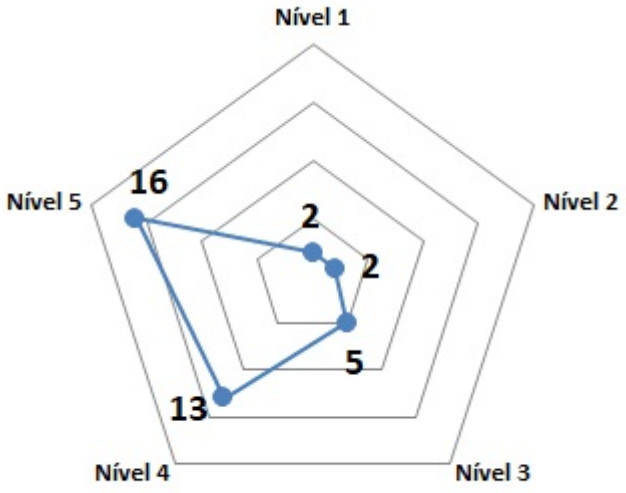


Figura 30 – Resultado relacionado à proibição de utilização de uma API

concentrou nos níveis mais elevados, 13 (34,2%) para o nível 4 e 16 (42,1%) para o nível 5. Vale ressaltar que 16 respostas para o nível 5 foi o maior valor que uma questão recebeu.

**Q6. Especificar a quantidade de IF's (desvios condicionais) que a solução deve ter?**

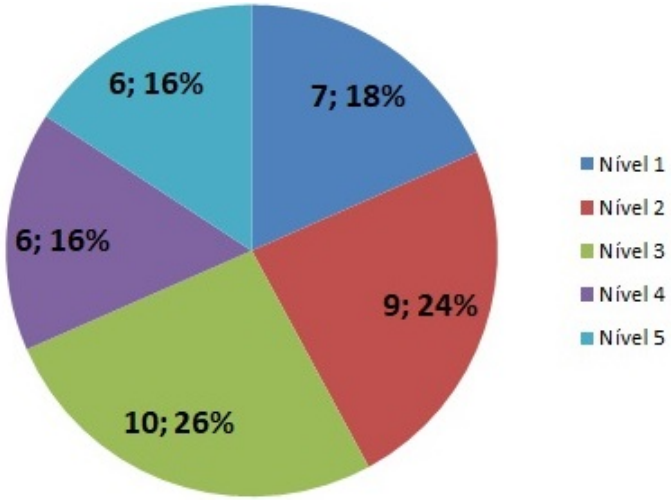


Figura 31 – Resultado relacionado a utilização de *ifs*

O mapeamento discutido no capítulo 4 mostrou que 29% das submissões que deveriam respeitar alguma restrição relacionada à construção *if* não respeitaram. Apesar disso, o resultado desta questão está bem distribuído entre todos os níveis. Isso significa que a utilização desta funcionalidade não é uma unanimidade entre os professores que participaram da pesquisa.

Considerando a figura 31, 7 (18,4%) professores disseram que não usariam a funcionalidade, 9 (23,7%) classificaram como nível 2, 10 (26,3%) como nível 3 e 12 professores optaram pelos níveis mais elevados de utilização, 6 (15,8%) como nível 4 e 6 (15,8%) como nível 5.

**Q7. Especificar a quantidade de condicionais do tipo SWITCH que a solução deve ter?**

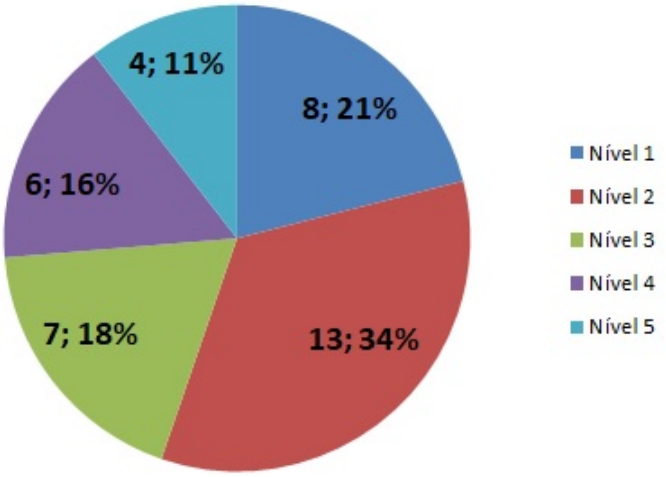


Figura 32 – Resultado relacionado ao comando *switch*

O resultado desta questão está levemente direcionado aos níveis mais baixos de utilização. Este resultado pode ser analisado em conjunto com a discussão feita no capítulo 4, na qual houve apenas um caso em que a construção *switch* foi levada em consideração. Além disso, esta construção não existe, por exemplo, na linguagem de programação Python, portanto quem utiliza esta linguagem não usaria esta funcionalidade. Diante disso, entende-se que apesar de estar presente nas ementas das disciplinas introdutórias de programação, não há uma grande necessidade de verificar se os alunos estão fazendo uso dela.

**Q8. Especificar a quantidade de laços do tipo FOR que a solução deve ter?**

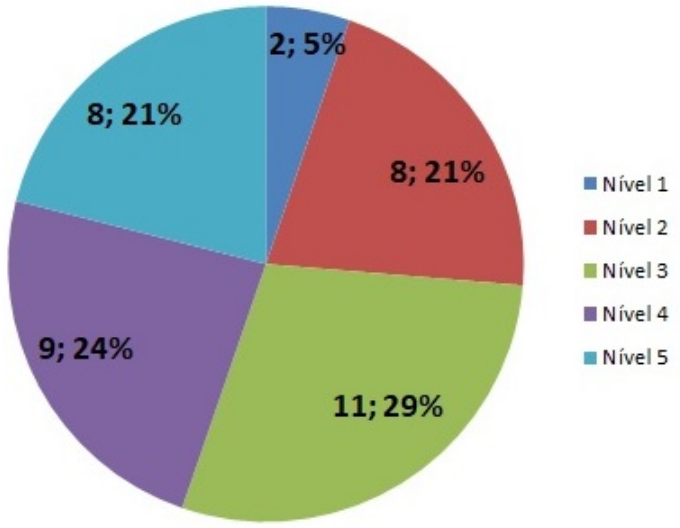


Figura 33 – Resultado relacionado ao comando *for*

Das três perguntas relacionas às estruturas de repetição (*for*, *while* e *do while*), a relacionada à construção *for* é a que possui mais classificações nos níveis mais elevados.

Ainda assim, o resultado não foi tão claro quanto as perguntas relacionadas a definições de funções. Conforme a figura 33, das 38 respostas, 10 (26,4%) estão direcionadas aos níveis mais baixos, 17 (44,8%) aos níveis mais altos e 11 (28,9%) professores optaram pelo nível 3.

**Q9. Especificar a quantidade de laços do tipo WHILE que a solução deve ter?**

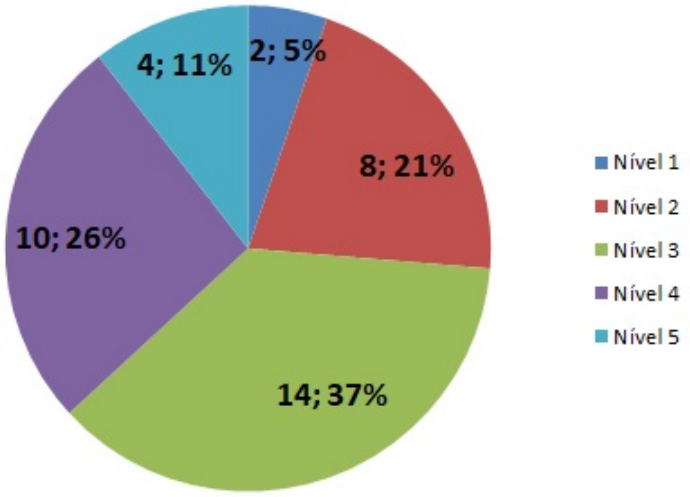


Figura 34 – Resultado relacionado ao comando *while*

Observando-se a figura 34, existe um claro equilíbrio entre as respostas dos professores. Das 38 respostas, 10 (26,4%) estão nos níveis mais baixos, 14 (36,8%) nos níveis mais altos e 14 (36,8%) professores optaram pelo nível 3.

**Q10. Especificar a quantidade de laços do tipo DO WHILE que a solução deve ter?**

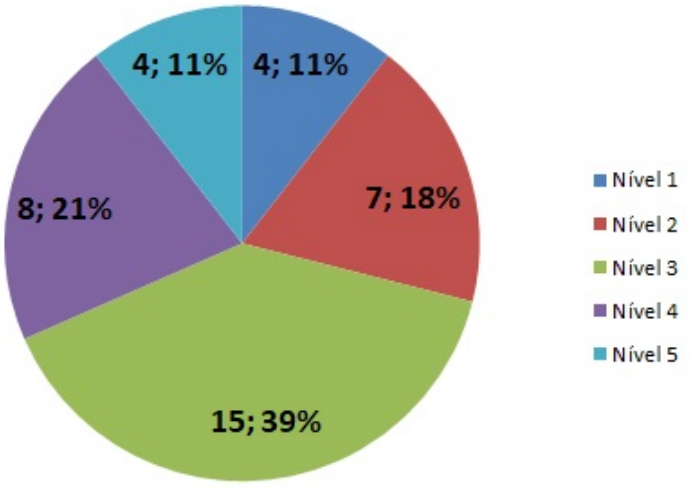


Figura 35 – Resultado relacionado ao comando *do while*

Assim como no comando *while*, neste caso também é notável um claro equilíbrio entre as respostas dos professores. Considerando a figura 35, das 38 respostas, 11 (28,9%) estão nos níveis mais baixos, 12 (31,6%) no níveis mais altos e 15 (39,5%) professores escolheram o nível 3.

**Q11. Especificar que é OBRIGATÓRIO utilizar ARRAY na solução?**

**Q12. Especificar que é PROIBIDO utilizar ARRAY na solução?**

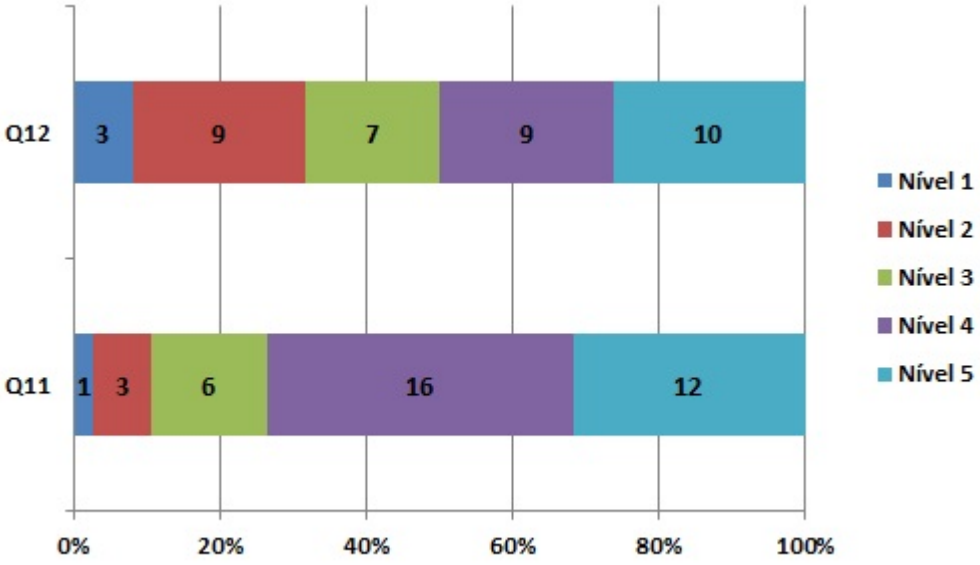


Figura 36 – Resultado relacionado à construção de *array*

As questões **Q11** e **Q12** estão relacionadas à utilização de *array* na solução. As respostas dos professores estão apresentadas na figura 36. Nela é possível notar que em ambas questões os níveis 4 e 5 foram os mais escolhidos pelos professores. Na questão **Q11**, os níveis 4 e 5 correspondem a 73% dos professores e na questão **Q12** os mesmos níveis correspondem a 50%.

Estes resultados podem ser analisados em conjunto com a discussão realizada no capítulo 4, na qual foi constatado que obrigar o aluno a utilizar vetores em suas soluções é a restrição mais utilizada com um percentual de 29% entre todas mapeadas. Diante disso, percebe-se que é necessário possuir uma ferramenta que inclua esta funcionalidade.

**Q13. Especificar que é OBRIGATÓRIO utilizar MATRIZ na solução?**

**Q14. Especificar que é PROIBIDO utilizar MATRIZ na solução?**

As questões **Q13** e **Q14** estão relacionadas à utilização de matriz na solução. As respostas dos professores estão apresentadas na figura 37. Assim como nas questões **Q11** e **Q12**, que versavam sobre array, neste caso os níveis 4 e 5 também foram os que mais se destacaram em ambas as questões.

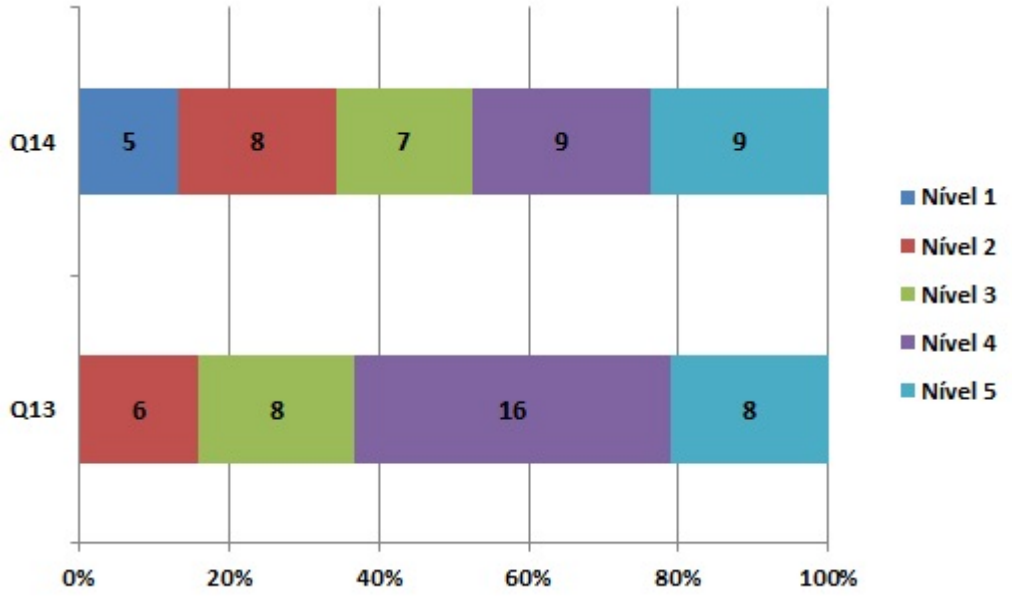


Figura 37 – Resultado relacionado à construção de *matriz*

Na questão **Q13** os níveis 4 e 5 correspondem a 63% e na questão **Q14** os mesmos níveis correspondem a 47% das respostas dos professores. Esses valores indicam que há tanto uma necessidade de obrigar quanto de proibir o aluno de utilizar matriz na solução. Para colaborar com este resultado, no capítulo 4 foi constatado que 15% de todas as restrições mapeadas estão relacionadas à utilização de matrizes.

**Q15. Especificar a utilização de tipos de variáveis?**

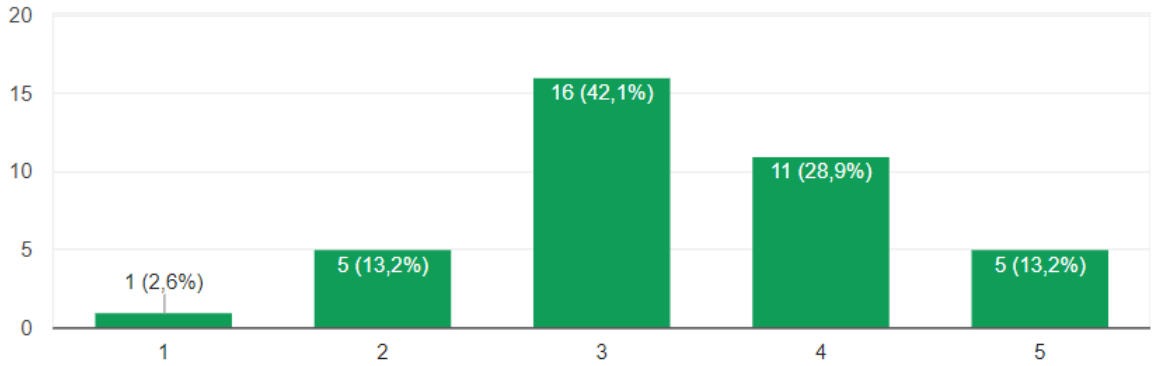


Figura 38 – Resultado relacionado à construção de variáveis tipadas

Observando-se a figura 38, é notável uma concentração de utilização no nível intermediário. Das 38 respostas, 6 (15,8%) estão nos níveis mais baixos, 16 (42,1%) estão nos níveis mais altos e 16 (42,1%) professores optaram pelo nível 3.

**Q16. Especificar que é OBRIGATÓRIO utilizar o conceito de LISTA na solução? (A resposta é opcional)**

**Q17. Especificar que é PROIBIDO utilizar o conceito de LISTA na solução? (A resposta é opcional)**

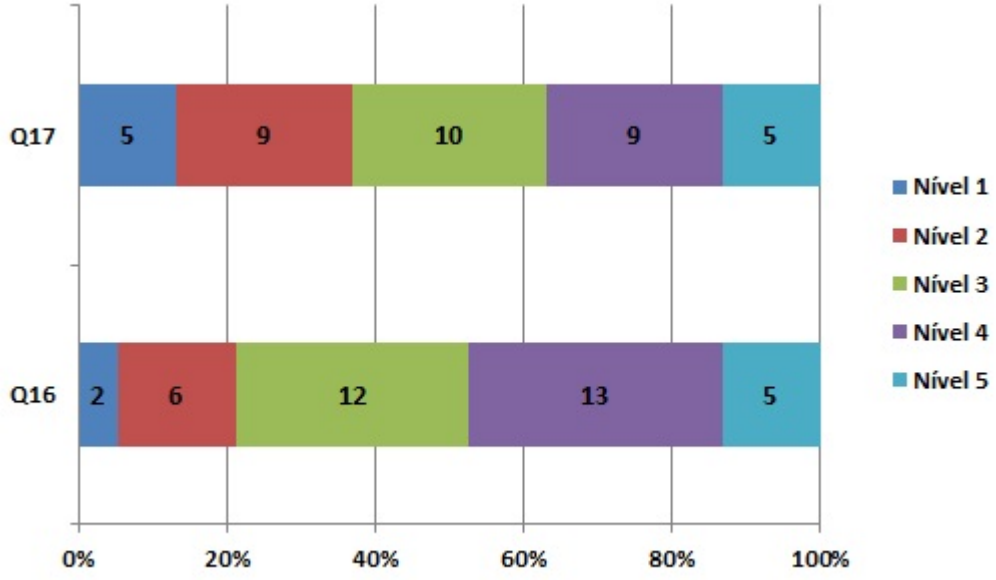


Figura 39 – Resultado relacionado à construção de lista

As questões **Q16** e **Q17** estão relacionadas à utilização de lista na solução. A figura 39 apresenta as respostas dos professores. O resultado dessas questões não mostra de forma tão evidente uma aceitação dos professores como aconteceu nas questões relacionadas a array e matriz.

Na questão **Q16**, 65% dos professores escolheram os níveis 3 ou 4, os mesmos níveis foram escolhidos por 50% dos professores na questão **Q17**. Esses números nos mostram que há uma inclinação maior para a obrigação de utilizar o conceito de Lista do que pela sua proibição.

**Q18. Especificar que é OBRIGATÓRIO utilizar o conceito de DICIONÁRIO na solução? (A resposta é opcional)**

**Q19. Especificar que é PROIBIDO utilizar o conceito de DICIONÁRIO na solução? (A resposta é opcional)**

A obrigatoriedade da utilização de dicionário não ficou clara de acordo com esta pesquisa. No capítulo 4 foi apresentado que 4% dos problemas que envolviam alguma restrição estavam relacionados ao uso de dicionários. Desta forma, esta funcionalidade não pode ser descartada, mas não possui o mesmo grau de relevância de outras como função e *array*.

A proibição de utilizar dicionários, assim como nas questões imediatamente anteriores, não está explícita. As respostas dos professores estão concentradas nos níveis 2, 3 e 4. Nenhum dos níveis extremos (mais baixo e mais alto) foi unanimidade entre os professores.

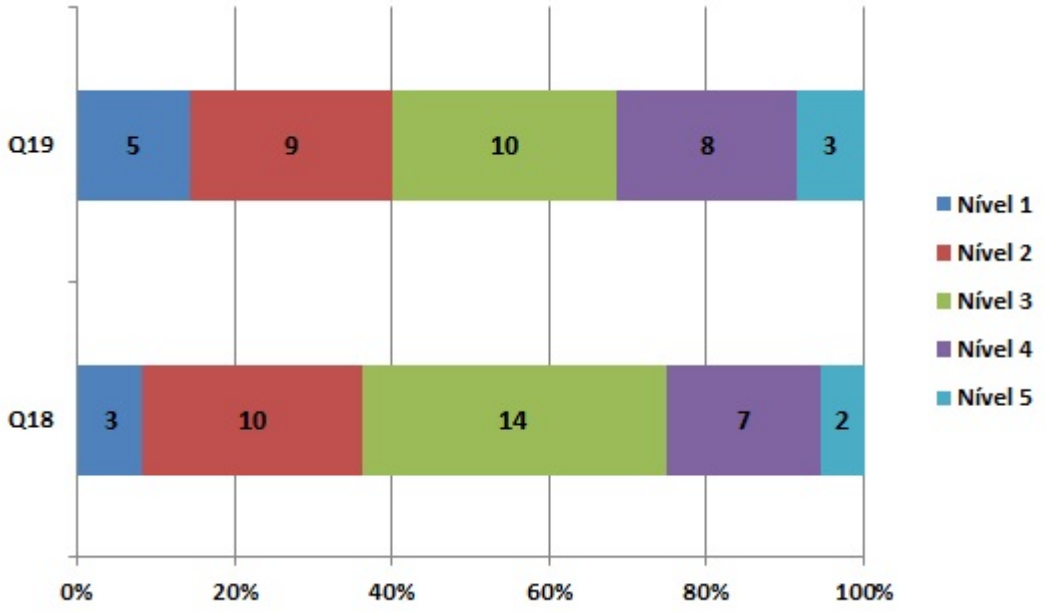


Figura 40 – Resultado relacionado à construção de dicionário

Isto indica que novamente há um equilíbrio entre as respostas e a funcionalidade não pode ser descartada.

**Q20. Especificar que é OBRIGATÓRIO utilizar o conceito de TUPLA na solução? (A resposta é opcional)**

**Q21. Especificar que é PROIBIDO utilizar o conceito de TUPLA na solução? (A resposta é opcional)**

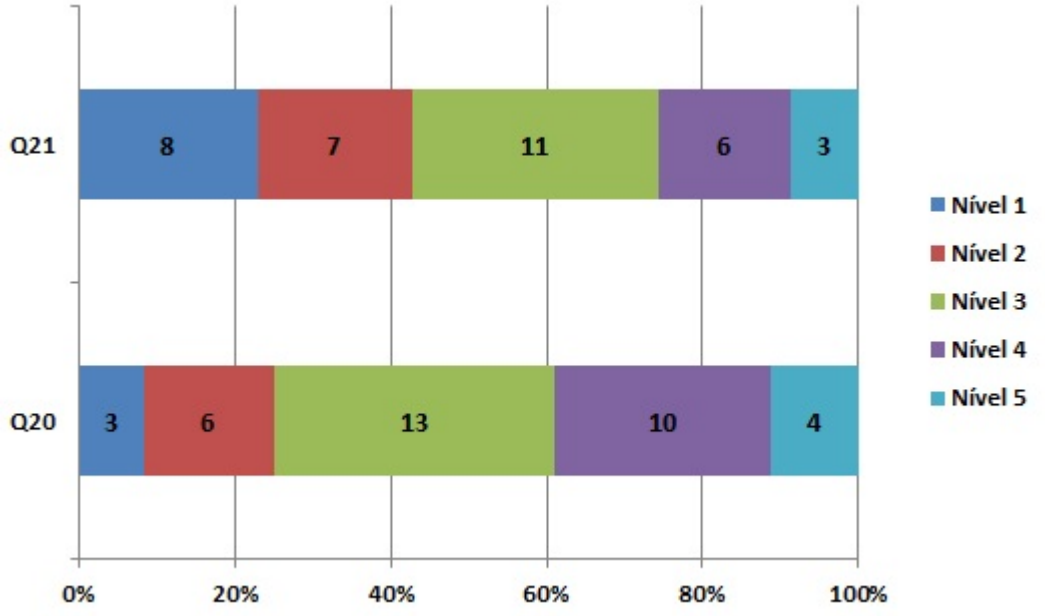


Figura 41 – Resultado relacionado à construção de tupla

As questões **Q20** e **Q21** tratam da utilização de tupla na solução dos alunos. A figura 41 traz as respostas dos professores.

Considerando as respostas para a questão **Q20** da figura 41, o nível intermediário recebeu o maior número de respostas, escolhido por 13 (36,1%) professores. Apenas 3 (8,3%) professores disseram que não usariam a funcionalidade, 6 (16,7%) escolheram o nível 2, 10 (27,8%) optaram pelo nível 4 e 4 (11,1%) pelo nível 5.

Não permitir que o aluno utilize uma tupla não foi uma funcionalidade que agradou a maioria dos participantes. Conforme a figura 41, das 38 respostas, 15 (41,6%) estão concentradas nos níveis mais baixos e apenas 9 (25%) nos níveis mais altos. O nível intermediário foi escolhido por 11 (30,6%) professores.

**Q22. Especificar que a solução DEVE UTILIZAR o comando CONTINUE?**

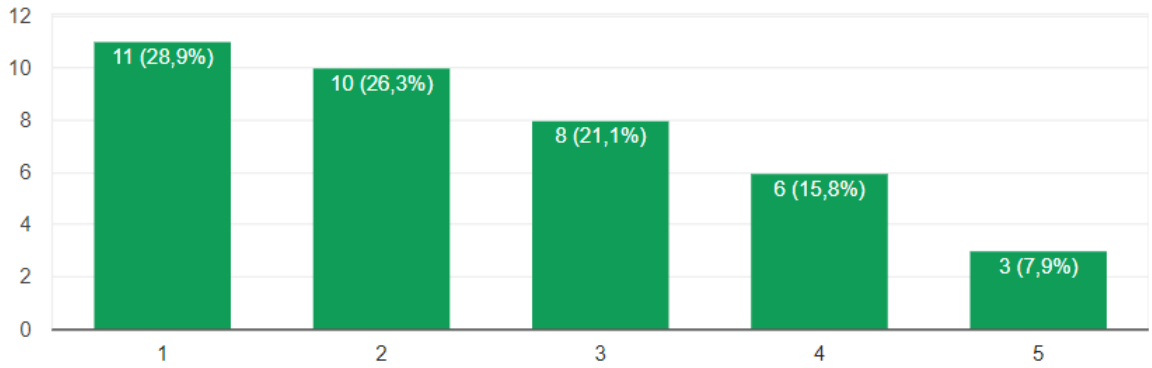


Figura 42 – Resultado relacionado ao comando *continue*

Observando-se a figura 42, é notável que os níveis mais baixos foram mais escolhidos do que os níveis mais altos. Isto dá indícios de que se fosse possível especificar que uma determinada solução deve utilizar o comando *continue*, esta funcionalidade não seria muito utilizada pelos professores que responderam o questionário.

O resultado desta questão também pode ter sido influenciado pela ideia que muitos profissionais têm a respeito dos comandos *continue* e *break*, que são comandos de desvio de fluxo. Esses tipos de comandos costumam ser comparados ao comando *go to*. O trabalho (DIJKSTRA, 1968) cita casos em que o comando *go to* foi demasiadamente utilizado e piorou a legibilidade do programa. Trabalhos como esse fundamentam as ideias de profissionais para a não utilização de comandos de desvio de fluxos.

**Q23. Especificar que a solução DEVE UTILIZAR o comando BREAK?**

Considerando a figura 43 também não houve uma clara aceitação da funcionalidade pelos professores participantes e um equilíbrio pode ser observado. Dos 38 professores, 8 (21,1%) disseram que não usariam o comando, outros 8 (21,1%) optaram pelo nível 2 de utilização, 10 (26,3%) pelo nível 3, 6 (15,8%) pelo nível 4 e outros 6 (15,8%) pelo nível máximo de utilização.

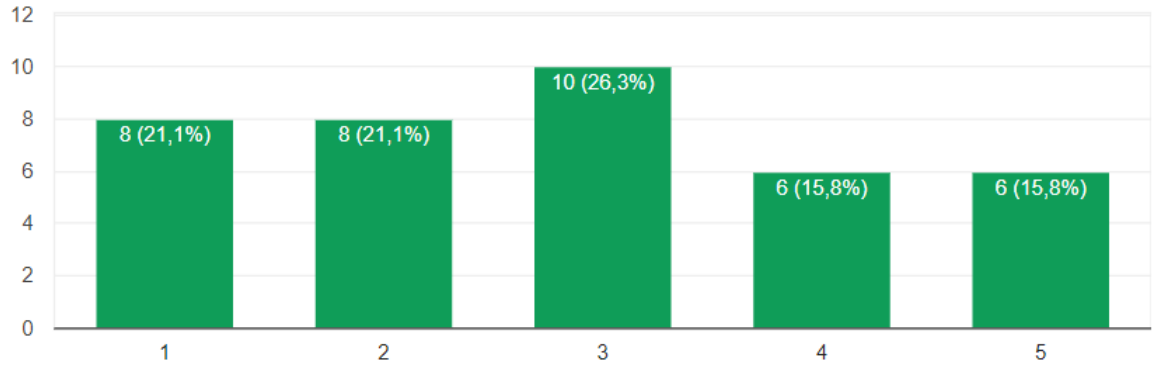


Figura 43 – Resultado relacionado ao comando *break*

**Q24. Especificar a quantidade máxima de linhas de código-fonte que a solução deve ter?**

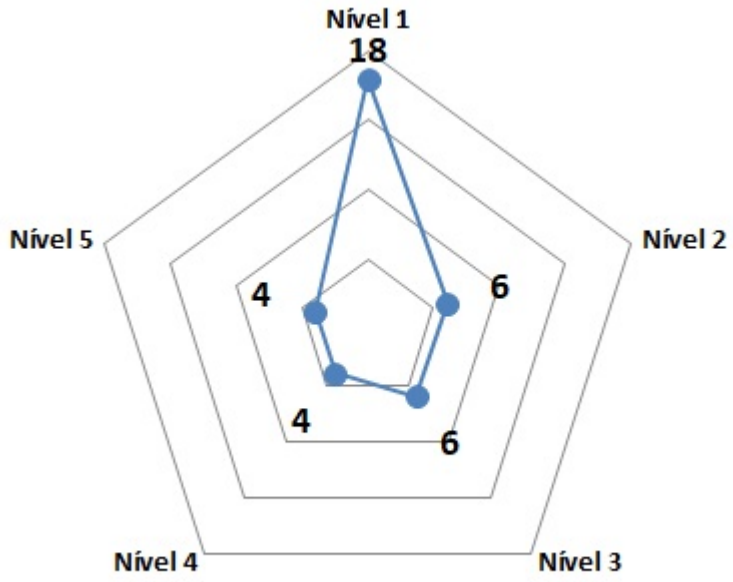


Figura 44 – Resultado relacionado a quantas linhas de código a solução deve ter

Esta questão possui o resultado mais claro de não utilização da funcionalidade. Conforme a figura 44, quase metade dos professores disseram que nunca usaria a funcionalidade (47%). Isto pode ser explicado pelo fato de que simplesmente o número de linhas de código não permite se chegar a conclusões a respeito do código-fonte. A quantidade de linhas de um programa pode ser alterada facilmente por meio de linhas em branco, comentários, estilo de programação e outras características inerentes ao aluno. Alguns alunos preferem utilizar uma linha específica para os *tokens* { e }, outros não fazem isto e escrevem o código com menos linhas por uma questão de estilo de programação.

**Q25. Quais outros aspectos do código-fonte você gostaria que fossem analisados no contexto de turmas introdutórias de programação?**

Respostas dos professores:

a) Limitar o uso máximo de memória durante a execução.

b) Indentação, nomes de variáveis significativos, interação adequada com o usuário.

c) Não usar variáveis globais. Usar parâmetros nas funções sem que for necessário acessar o conteúdo de uma variável externa a função.

d) identação e estruturação do código.

e) É importante checar o espaçamento do programa e se possui comentários.

f) Limitar / Exigir o uso de variaveis globais

g) Nomes de variáveis

h) Classes/Registros/estruturas (struct e class)

i) A maioria das vezes, apenas damos o problema e o discente deve construir uma solução. Neste processo, esperamos que ele avalie e aplique a estrutura mais adequada (seja de controle de fluxo ou de dados). Portanto, considerem uma excelente ferramenta para a análise de correção dos códigos com a checkpoints definidos pelo professor... por exemplo, além do habitual critério de saída esperada, analisar se a solução usa um for [invés de while] e se variável usada para saída do tipo x.

j) A quantidade máxima de estruturas aninhadas; Número máximo de variáveis usadas;

k) Especificar que a solução deve utilizar estruturas (struct); Um pouco mais dificil, mas poderia ser feito: Especificar a complexidade da solução. Por exemplo, colocar um limite superior para a quantidade de operações. Ordenação por exemplo, se o aluno não usar um algoritmo n log n ele nao vai conseguir cumprir essa meta.

l) Identação e nomes das variavéis.

m) indentação e organização em geral do código

n) Se estão usando a linguagem abordada em sala de aula e não irá linguagem qualquer

o) Poucos níveis de aninhamento de condicionais e loops (ou mais geralmente, low cyclomatic complexity).

   Funções e procedimentos pequenos (poucas linhas).

   O programa é dividido em funções e procedimentos.

   Funções são puras (não realizam efeitos colaterais).

   Nomes adequados Segue padrões de codificação (usa indentação, nomes seguem um padrão).

p) Comparar códigos entre alunos, isto é, se houve cópia entre eles.

q) O consumo de memória e tempos de execução das soluçoes seria interessane. Penso que seria mais interessante que uma ferramenta sugerisse alternativas e nao especifique tao fortemente os elementos que uma soluçao poderia ter para ser considerada correta. Assim seria possível que o estudante começe a identificar casos em que

o uso de determinadas abordagens e estruturas seriam mais convenientes do que outras. Afinal lidamos com uma geraçao com serios comprometimentos no requisito criatividade, e programar é um exercício de criatividade.

r) A complexidade ciclomática da solução desenvolvida e a cobertura de testes. Testes de código deveriam ser ensinados desdes as turmas introdutórias. A técnica facilitaria inclusive a correção pelos professores que poderiam passar questões já com os testes associados para cada questão solicitada.

**Q26. Qual a sua formação?**

A formação dos professores variou entre os seguintes cursos: Ciência da Computação (26), Informática (2), Sistemas de Informação (1), Engenharia da Computação (cursando) (1), Engenharia Civil (1), Computação e Educação (1), Análise de Sistemas (1), Licenciatura em Computação (1), Licenciatura em Informática (1), Computação/Engenharia (1), Matemática e Informática (1), Mestrado em Computação (não informou o curso).

**Q27. Qual a sua maior formação de Pós-Graduação?**

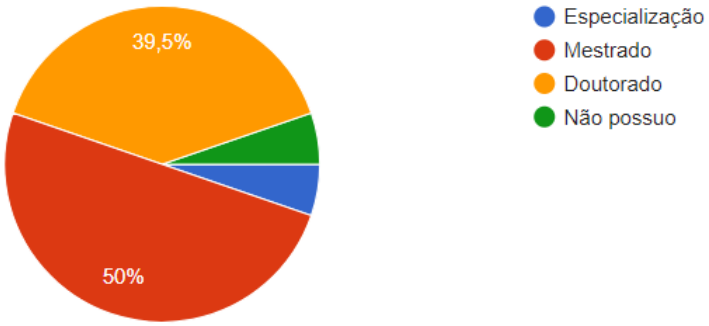


Figura 45 – Maior formação de pós-graduação

A metade dos participantes possui o Mestrado como sua maior formação de pós-graduação, 39.5% possuem Doutorado, 2.25% possuem Especialização e 2.25% não possui pós-graduação.

**Q28. Para quantas turmas você já lecionou Introdução à programação?**

A maior parte dos professores (52.6%) já lecionou para mais de 10 turmas introdutórias de programação, isso nos diz que a pesquisa contou com profissionais experientes na área. 21.1% dos professores apenas lecionaram para até 3 turmas, 10.5% entre 7 e 9 turmas e 15.8% entre 4 e 6 turmas.

### 6.2.2 Comparação com a abordagem proposta

Das funcionalidades sondadas neste questionário, as que mais se destacaram foram:

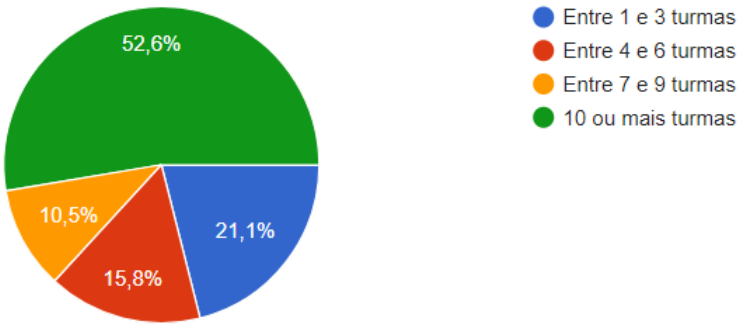


Figura 46 – Experiência no ensino de programação

- Criação de uma função com um determinado nome e quantidade de parâmetros.
- Criação de uma função com ou sem retorno.
- Criação de uma função recursiva.
- Utilização de uma função da API da linguagem de programação.
- Não utilização de uma função da API da linguagem de programação.
- Utilização de array na solução.
- Utilização de matriz na solução.

O quadro 6.4 apresenta um resumo dos principais tópicos abordados no questionário, na base do The Huxley e na Abordagem Proposta. Na coluna Questionário, apenas os tópicos mais relevantes para os professores que participaram da pesquisa são apresentados. De acordo com este quadro, é notável que há um alinhamento do que foi proposto por este trabalho com os resultados do questionário e com a base do The Huxley.

Entretanto, este alinhamento não pode ser considerado completo porque há tópicos encontrados na base do The Huxley que não estão contemplados na Abordadem Proposta, como: *switch* e laço *do while*. Conforme já foi explicado no capítulo 4, esses tópicos não foram abordados neste trabalho porque o mesmo possui o objetivo de desenvolver uma abordagem unificada em que fosse possível fazer inicialmente um mapeamento das restrições para as linguagens de programação Java e Python. Como a linguagem Python não possui os conceitos de *switch* e `do while`, esses assuntos não foram abordados.

A Abordagem Proposta ainda possui funcionalidades que não foram encontradas em nenhuma das duas fontes (Questionário e Base do The Huxley), como: blocos vazios, utilização (ou proibição de utilização) de vetor/matriz e lista. A inclusão desses tópicos foi baseada nos trabalhos relacionados (PELZ, 2014), (HODECKER, 2014) e em experiências em sala de aula.

| **Questionário** | **Base do The Huxley** | **Abordagem Proposta** |
|---|---|---|
| Função | Função | Função |
| Array | Array | Array |
| Matriz | Matriz | IF |
| Função da API | Laço | While |
| - | If | For |
| - | Dicionário | Blocos Vazios |
| - | Switch | Lista |
| - | - | Dicionário |
| - | - | Função da API |

Quadro 6.4 – Resumo dos tópicos abordados no questionário, na base do The Huxley e na Abordagem Proposta

# 7

# Conclusão

Este trabalho apresentou uma abordagem para especificar e checar restrições de código-fonte para soluções de problemas de programação. Para o desenvolvimento desta abordagem, foi necessário verificar quais restrições de código-fonte eram mais frequentes. Esta verificação foi feita através de uma análise da base de dados do The Huxley.

Por meio da análise das submissões de todos os problemas criados até o final de 2016 com nível adequado para alunos de introdução à programação, foi possível perceber quais são as restrições frequentemente exigidas pelos professores e quais são as menos respeitadas pelos alunos. As restrições relacionadas à utilização/proibição de array/matriz e funções recursivas ou não são as mais utilizadas pelos professores. A restrição de criação e utilização de funções recursivas foi a menos respeitada pelos alunos. Além de servir como base para o desenvolvimento da abordagem, esta análise é considerada fundamental e constitui uma importante contribuição porque em nenhum trabalho abordado nesta pesquisa foi encontrado um levantamento como este. A motivação utilizada pelos trabalhos relacionados não envolveu um levantamento de uma extensa base de dados, como o realizado nesta pesquisa sobre a base do The Huxley. Os outros trabalhos utilizaram apenas simples exemplos de como o aluno poderia não seguir as restrições estabelecidas pelos professores e ainda assim conseguir a resposta correta para o problema.

Após o desenvolvimento da abordagem, foi realizada uma comparação com outros três trabalhos: Scheme-robo (SAIKKONEN; MALMI; KORHONEN, 2001), Projekt Tomo (JERSE; LOKAR, 2016) e Portugol Studio (PELZ, 2014), (HODECKER, 2014). Esta comparação utilizou os seguintes critérios: facilidade de uso, nível de flexibilidade e abordagem unificada. De acordo com os critérios estabelecidos, foi notável que a abordagem desenvolvida neste trabalho possui o melhor *trade-off*, pois, apesar de não ter um nível de flexibilidade alto como Scheme-robo e Projekt Tomo ou baixo como Portugol Studio, oferece um nível de flexibilidade adequado em relação à análise da base do The Huxley. Este trabalho também é o único a utilizar uma abordagem unificada que foi projetada para funcionar da mesma forma em diversas linguagens de

programação. Além disso, como permite que uma interface gráfica seja utilizada para especificar e checar as restrições, é considerado de fácil utilização, como o Portugol Studio.

Além de uma comparação com os principais trabalhos da literatura, este trabalho foi validado pelos resultados de um *survey* aplicado aos professores de programação cadastrados na lista de *e-mails* da SBC. Neste *survey*, os professores responderam quais funcionalidades usariam se tivessem à disposição uma ferramenta de análise estática. As funcionalidades mais escolhidas pelos professores foram: criação de funções recursivas ou não, utilização de array/matriz, utilização de determinada função da API e proibição de utilização de determinada função da API. Todas essas funcionalidades estavam no escopo inicial da nossa abordagem e foram implementadas. Deste forma, nossa abordagem está alinhada com o que a maioria dos professores que responderam ao *survey* esperam de uma ferramenta de análise estática.

Este trabalho possui três principais contribuições:

- Uma abordagem para avaliação estática de exercícios de programação;
- Uma análise da base de dados de The Huxley com o propósito de descobrir quais são as principais restrições utilizadas pelos professores e se estas são seguidas pelos alunos;
- A aplicação de um *survey* com o objetivo de descobrir quais são as funcionalidades de avaliação estática mais importantes do ponto de vista dos professores.

De acordo com o que foi apresentado nesta dissertação, percebe-se que o objetivo geral deste trabalho foi alcançado, pois o mesmo realizou uma melhoria no processo de correção de problemas utilizado pelos juízes *on-line*. Além disso, alcançou os objetivos específicos por meio das contribuições já mencionadas.

## 7.1 Trabalhos futuros

Como trabalho futuro, pretende-se analisar as respostas obtidas pelo *survey* apresentado no capítulo 6.1.4.3 para criar novas formas úteis de avaliar estaticamente uma solução para um problema de programação. Esta avaliação pode ser derivada de métricas de software que envolvam complexidade ciclomática, comentários de código-fonte, nomes de variáveis, tamanho de funções e outros aspectos relevantes.

Também é pretendido seguir os mesmos passos trilhados nesta pesquisa para que novas linguagens de programação sejam suportadas pela nossa abordagem, como C e C++.

# Referências


ALA-MUTKA, K. M. A survey of automated assessment approaches for programming assignments. *Computer science education*, Taylor & Francis, v. 15, n. 2, p. 83–102, 2005. Citado na página 29.

ALVES, F. P.; JAQUES, P. Um ambiente virtual com feedback personalizado para apoio a disciplinas de programação. In: *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*. [S.l.: s.n.], 2014. v. 3, n. 1, p. 51. Citado na página 27.

BEZ, J. L.; TONIN, N. A.; RODEGHERI, P. R. URI Online Judge Academic: A Tool for Algorithms and Programming Classes. *The 9th International Conference on Computer Science & Education (ICCSE 2014)*, n. Iccse, p. 149–152, 2014. Citado na página 26.

CARDOSO, A. L. M. d. S. Construção e difusão colaborativa do conhecimento: uma experiência construtivista de educação em um ambiente virtual de aprendizagem. 2010. Citado na página 19.

DIERBACH, C. Python as a first programming language. *Journal of Computing Sciences in Colleges*, Consortium for Computing Sciences in Colleges, v. 29, n. 6, p. 153–154, 2014. Citado na página 32.

DIJKSTRA, E. W. A case against the {GO TO} statement. 1968. Citado na página 86.

DWARS, M. *Java Antlr Grammar*. 2013. Disponível em: <https://github.com/antlr/grammars-v4/tree/master/java>. Acesso em: 22 nov. 2017. Citado na página 48.

E-PROINFO. 2016. Disponível em: <http://portal.mec.gov.br/expansao-da-rede-federal/114-conhecaomec-1447013193/sistemas-do-mec-88168494/138-e-proinfo>. Acesso em: 13 nov. 2016. Citado na página 19.

ERICH, G. et al. *Padrões de Projeto: Soluções reutilizáveis de software orientado a objeto.* [S.l.]: Bookman, 2000. Citado na página 51.

EVERETT, T. *Python 3 parser*. 2014. Disponível em: <https://github.com/antlr/grammars-v4/tree/master/python3>. Acesso em: 22 nov. 2017. Citado na página 48.

FEEPER. *FEEPER: Um ambiente web para apoio ao ensino de programação*. 2016. Disponível em: <http://feeper.unisinos.br/>. Acesso em: 13 nov. 2016. Citado na página 20.

FRANCISCO, R. E.; JúNIOR, C. P.; AMBROSIO, A. Juiz online no ensino de programação introdutória - uma revisão sistemática da literatura. 10 2016. Citado na página 66.

GIMÉNEZ, O.; PETIT, J.; ROURA, S. Jutge.org: An Educational Programming Judge. In: *Special Interest Group on Computer Science Education (SIGCSE 2012)*. [S.l.: s.n.], 2012. p. 445–450. ISBN 9781450310987. Citado 2 vezes nas páginas 26 e 27.

HODECKER, A. *Aprimoramento e avaliação do corretor de questões do Portugol Studio*. Tese (Trabalho técnico-científico de conclusão de curso) — Universidade do Vale do Itajaí, 2014. Citado 5 vezes nas páginas 12, 70, 71, 90 e 92.

HUXLEY, T. *Decrescente 3*. 2011. Disponível em: <http://thehuxley.com/problem/41>. Acesso em: 11 nov. 2017. Citado na página 37.

HUXLEY, T. *O Maior*. 2011. Disponível em: <http://thehuxley.com/problem/293>. Acesso em: 11 nov. 2017. Citado na página 38.

HUXLEY., T. *Escola de Música*. 2014. Disponível em: <https://www.thehuxley.com/problem/404>. Acesso em: 24 nov. 2016. Citado na página 20.

JERSE, G.; LOKAR, M. *Using system for automatic assessment Projekt Tomo in learning and teaching numerical methods*. 2016. Citado 2 vezes nas páginas 68 e 92.

JUDGE, S. O. 2016. Disponível em: <http://www.spoj.com/>. Acesso em: 13 nov. 2016. Citado na página 20.

MARCOLINO, A.; BARBOSA, E. F. Softwares educacionais para o ensino de programaçao: Um mapeamento sistemático. In: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. [S.l.: s.n.], 2015. v. 26, n. 1, p. 190. Citado na página 19.

MOODLE. 2016. Disponível em: <https://docs.moodle.org/31/en/About_Moodle>. Acesso em: 13 nov. 2016. Citado na página 19.

MORCOURT., V. R. *Base e expoente*. 2016. Disponível em: <https://www.thehuxley.com/problem/902>. Acesso em: 24 nov. 2016. Citado na página 22.

OLIVEIRA, A. S.; MOTA, L. da C.; OLIVEIRA, A. A. de. Uso de ambientes virtuais de aprendizagem como suporte ao ensino de programação: uma revisão sistemática. *Interfaces Científicas-Exatas e Tecnológicas*, v. 1, n. 3, p. 9–22, 2015. Citado 2 vezes nas páginas 19 e 26.

ORACLE. *Class ArrayList<E>*. 1996. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html>. Acesso em: 22 nov. 2017. Citado na página 42.

ORACLE. *Class ArrayList<E>*. 1998. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>. Acesso em: 22 nov. 2017. Citado na página 42.

ORACLE. *Class LinkedList<E>*. 1998. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>. Acesso em: 22 nov. 2017. Citado na página 42.

PAES, R. de B. et al. Ferramenta para a avaliação de aprendizado de alunos em programação de computadores. In: *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*. [S.l.: s.n.], 2013. v. 2, n. 1. Citado 3 vezes nas páginas 20, 26 e 27.

PARR, T. *The definitive ANTLR reference*. [S.l.]: Pragmatic Bookshelf, 2007. Citado na página 70.

PARR, T. *The definitive ANTLR 4 reference*. [S.l.]: Pragmatic Bookshelf, 2013. Citado 2 vezes nas páginas 24 e 48.

PELZ, F. D. *Um gerador de dicas para guiar novatos na aprendizagem de programação*. Tese (Dissertação (Mestrado em Computação Aplicada)) — Universidade do Vale do Itajaí, 2014. Citado 4 vezes nas páginas 12, 70, 90 e 92.

PELZ, F. D.; JESUS, E. A. de; RAABE, A. L. Um mecanismo para correção automática de exercícios práticos de programação introdutória. In: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. [S.l.: s.n.], 2012. v. 23, n. 1. Citado na página 70.

PELZ, F. D.; RAABE, A. L. A. Análise do feedback fornecido por corretores de algoritmos com propósito educacional. *Conferencias LACLO*, v. 4, n. 1, 2013. Citado na página 70.

PYTHON. *The Python Standard Library - Abstract Syntax Trees*. 2011. Disponível em: <https://docs.python.org/2/library/ast.html>. Acesso em: 10 fev. 2017. Citado na página 69.

RAHMAN, K. A.; NORDIN, M. J. A review on the static analysis approach in the automated programming assessment systems. 2007. Citado 2 vezes nas páginas 29 e 66.

SAIKKONEN, R.; MALMI, L.; KORHONEN, A. Fully automatic assessment of programming exercises. In: ACM. *ACM Sigcse Bulletin*. [S.l.], 2001. v. 33, n. 3, p. 133–136. Citado 3 vezes nas páginas 30, 66 e 92.

SHEIN, E. Python for beginners. *Communications of the ACM*, ACM, v. 58, n. 3, p. 19–21, 2015. Citado na página 32.

SOMMERVILLE, I. *Engenharia de Software-9ª Edição (2011)*. [S.l.]: Ed Person Education, 2011. Citado na página 29.

STUDIO, P. *Portugol Studio*. 2017. Disponível em: <http://lite.acad.univali.br/portugol/>. Acesso em: 10 fev. 2017. Citado 2 vezes nas páginas 30 e 69.

SUN, H.; Bofang Li, M. J. YOJ: An online judge system designed for programming courses. In: *2014 9th International Conference on Computer Science & Education*. IEEE, 2014. p. 812–816. ISBN 978-1-4799-2951-1. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6926575>. Citado na página 26.

URI. *Problems and Contests*. 2016. Disponível em: <https://www.urionlinejudge.com.br/>. Acesso em: 13 nov. 2016. Citado na página 20.

UVA. 2016. Disponível em: <https://uva.onlinejudge.org/>. Acesso em: 13 nov. 2016. Citado na página 20.

WU, J.; Shuangping Chen, R. Y. Development and application of online judge system. In: *2012 International Symposium on Information Technologies in Medicine and Education*. IEEE, 2012. v. 1, p. 83–86. ISBN 978-1-4673-2108-2. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6291253>. Citado na página 26.

# A

# Problemas difíceis

Este apêndice apresenta os problemas que no dia 19/09/2017 estavam com classificação abaixo de 4, no entanto, pelos motivos apresentados no quadro A.1 foram considerados difíceis para alunos de introdução à programação.

| Nome do problema | Motivo para não inclusão na análise |
|---|---|
| Dama | Problema de maratona de programação |
| Revisão de Contrato | Problema de maratona de programação |
| Leitura Ótica | Problema de maratona de programação |
| Interior da área | Problema de geometria computacional |
| Macarronada | Problema de geometria computacional |
| Guarda costeira | Problema de maratona de programação |
| Foco | Problema de maratona de programação |
| Álbum | Problema de geometria computacional |
| Balanceamento de Parênteses | Estrutura de dados Pilha |
| Todos os subconjuntos | Backtracking |
| Calculadora de Expressões | Estrutura de dados Pilha |
| A permutação | Típico de maratona de programação |
| Insertion Sort | Estrutura de dados Insertion Sort |
| Selection Sort | Estrutura de dados Selection Sort |
| Árvore de Busca Binária | Estrutura de dados Árvore |
| Profundidade de um nó em uma árvore binária | Estrutura de dados Árvore |
| Pilhas com Listas | Estrutura de dados Pilha |
| Ciclos | Grafos |
| Quicksort | Estrutura de dados Quicksort |
| Empilhando Dominó | Problema do Codeforces para maratona |
| Ambrósio e a caixa de moedas | Programação dinâmica |

| | |
|---|---|
| Divisão da Nlogônia | Problema de geometria computacional |
| Alarme despertador | Problema de maratona de programação |
| Jacutingas vs Jaburus | Problema de maratona de programação |
| Itens de Caruaru | Problema de maratona de programação |
| Jarathus, o Conquistador | Problema de maratona de programação |
| Inserção em Árvores de Busca Binária | Estrutura de dados Árvore |
| Mas que árvore estranha | Estrutura de dados Árvore |
| Esta solução do Sudoku está correta? | Problema do Spoj para maratona |
| Mochila | Problema da mochila |
| Compotência | Problema do Erbase para maratona |
| Willy Ecológico | Envolve busca binária |
| Maior Sequência | Estrutura de dados lista encadeada |
| Mozilla Spring Camp | Estrutura de dados pilha |
| Somatório em um array circular | Estrutura de dados lista circular |
| Batata quente | Problema de maratona de programação |
| A Floresta de Binágoras | Estrutura de dados Árvore |
| Troco | Programação dinâmica |
| Torre | Problema da OBI para maratona |
| Campo de Minhocas | Problema da OBI para maratona |
| Nlogonia e suas estradas | Grafos |
| Nota de História | Programação dinâmica |
| Soma de grupos divisíveis por D | Programação dinâmica |
| Remoção de strings | Programação dinâmica |
| Cavalos do Xadrez | Backtracking |
| Resolvendo o Sistema | Backtracking |
| P2, vamos lá! | Estrutura de dados Árvore |
| A soma de todos os medos | Programação dinâmica |

Quadro A.1 – Problemas com nível abaixo de 4, mas considerados difíceis

# B

# Manual do Analisador Estático

Este apêndice apresenta como deve ser feita a comunicação de uma aplicação cliente com o Analisador Estático.

## B.1 Tipos de restrições

Esta seção descreve como deve ser formatado o arquivo *Json* com as restrições que servirão de entrada para o Analisador Estático. Para utilizar algum tipo de restrição é necessário especificar os valores de **todos os atributos** apresentados para tal restrição.

As próximas subseções possuem o seguinte formato: um quadro que informa quais os atributos e seus respectivos domínios de valores, um exemplo de utilização da restrição e um breve comentário a respeito do exemplo utilizado.

### B.1.1 Desvios Condicionais (*If*)

| Atributo | Valor |
|---|---|
| minimo | Um **inteiro** com a quantidade mínima de desvios condicionais que a solução deve ter. |
| maximo | Um **inteiro** com a quantidade máxima de desvios condicionais que a solução deve ter. |

Quadro B.1 – Atributos e valores para a restrição desvios condicionais.

Exemplo:

```
1 {"restricaoIf":
2   {"minimo":1,
3   "maximo":5}
4 }
```

Listagem B.1 – Especificação da restrição desvios condicionais.

O exemplo da listagem B.1 especifica que a solução deve ter no mínimo 1 e no máximo 5 desvios condicionais.

### B.1.2 Especificação de Funções

| **Atributo** | **Valor** |
|---|---|
| nome | Uma **string** com o nome da função . |
| retorno | **true** ou **false** indicando se a função deve ter retorno ou não. |
| recursiva | **true** ou **false** indicando se a função deve ser recursiva ou não. |
| qtdParametros | Um **inteiro** que representa a quantidade de parâmetros da função. |

Quadro B.2 – Atributos e valores para a restrição de funções.

Exemplo:

```
{"restricaoFuncoes":
  [
    {"nome":"somar",
    "retorno":true,
    "recursiva":false,
    "qtdParametros":2},

    {"nome":"subtrair",
    "retorno":true,
    "recursiva":false,
    "qtdParametros":2}
  ]
}
```

Listagem B.2 – Especificação de duas funções.

A listagem B.2 especifica que a solução deve criar e invocar duas funções, um chamada chamada somar, com retorno, não recursiva e com dois parâmetros e outra chamada subtrair, com retorno, não recursiva e com dois parâmetros.

### B.1.3 While

| **Atributo** | **Valor** |
|---|---|
| minimo | Um **inteiro** com a quantidade mínima de while que a solução deve ter . |
| maximo | Um **inteiro** com a quantidade máxima de while que a solução deve ter . |

Quadro B.3 – Atributos e valores para a restrição while.

Exemplo:

```
{"restricaoWhile":
  {"minimo":2,
  "maximo":5}
}
```

Listagem B.3 – Especificação da restrição while.

A listagem B.3 especifica que a solução deve ter no mínimo 2 e no máximo 5 construções *while*. Neste caso não é verificado se as construções estão aninhadas ou não.

### B.1.4 For

| Atributo | Valor |
|---|---|
| minimo | Um **inteiro** com a quantidade mínima de for que a solução deve ter . |
| maximo | Um **inteiro** com a quantidade máxima de for que a solução deve ter . |

Quadro B.4 – Atributos e valores para a restrição for.

Exemplo:

```
{"restricaoFor":
  {"minimo":0,
  "maximo":2}
}
```

Listagem B.4 – Especificação da restrição for

A listagem B.4 especifica que a solução deve ter no máximo 2 comandos *for*. Assim como na construção *while*, nesta também não é verificado se elas estão aninhadas ou não.

### B.1.5 Lista

| Atributo | Valor |
|---|---|
| possuiLista | **true** se a solução necessitar da declaração de lista ou **false** se for proibido. |

Quadro B.5 – Atributos e valores para a restrição lista.

Exemplo:

```
{"restricaoLista":{
  "possuiLista":true}
}
```

Listagem B.5 – Especificação de restrição que exige a declaração de uma lista.

A listagem B.5 especifica que a submissão deve declarar uma lista para resolver o problema.

### B.1.6 Dicionário

| Atributo | Valor |
|---|---|
| possuiDicionario | **true** se a solução necessitar da declaração de um dicionário ou **false** se for proibido. |

Quadro B.6 – Atributos e valores para a restrição dicionário.

Exemplo:

```
1 {"restricaoDicionario":{
2   "possuiDicionario":true}
3 }
```

Listagem B.6 – Especificação de restrição que exige que a solução possua a declaração de um dicionário.

Assim como nas restrições de lista, neste caso apenas um atributo é necessário. O exemplo da listagem B.6 aponta que é obrigatório declarar um dicionário para resolver o problema.

### B.1.7 Vetor/Matriz

| Atributo | Valor |
|---|---|
| possuiVetor | **true** se a solução necessitar da declaração de um vetor/matriz ou **false** se for proibido. |

Quadro B.7 – Atributos e valores para a restrição vetor/matriz.

Exemplo:

```
1 {"restricaoVetor":{
2   "possuiVetor":true}
3 }
```

Listagem B.7 – Especificação de restrição que exige que a solução possua a declaração de um vetor/matriz.

O exemplo da listagem B.7 especifica que é obrigatório declarar um vetor/matriz para resolver o problema.

### B.1.8 Blocos vazios

| Atributo | Valor |
|---|---|
| possuiBlocosVazios | **false** se for proibido utilizar blocos de código vazios. |

Quadro B.8 – Atributos e valores para a restrição blocos vazios.

Exemplo:

```
{"restricaoBlocosVazios":
  {"possuiBlocosVazios":false}
}
```

Listagem B.8 – Especificação de restrição que impede a solução de ter blocos vazios.

Neste caso apenas o valor *false* é admitido para o único atributo. Se o professor não quiser checar os blocos vazios da solução ele pode simplesmente não utilizar esta restrição. O exemplo da listagem B.8 aponta que a solução não deve conter blocos vazios.

### B.1.9 Chamada de função

| **Atributo** | **Valor** |
|---|---|
| nome | Uma **string** com o nome de uma função que deve ser invocada no código-fonte. |
| qtdParametros | Um **inteiro** com a quantidade de parâmetros que a função invocada deve ter. |
| utilizacao | **true** se for obrigatório invocar a função ou **false** se for proibido. |

Quadro B.9 – Atributos e valores para a restrição chamada de função.

Exemplo:

```
{"restricaoFuncaoChamada":{
  "nome":"max",
  "qtdParametros":1,
  "utilizacao":false}
}
```

Listagem B.9 – Especificação de restrição chamada de função.

O exemplo da listagem B.9 aponta que é proibido invocar a função *max* que recebe apenas um parâmetro para solucionar o problema.

### B.1.10 Exemplo com vários tipos

Nesta subseção é apresentado um exemplo com a especificação de várias restrições.

```
{
  "restricaoWhile":{
    "minimo":0,
    "maximo":1},

  "restricaoIf":{
    "minimo":0,
    "maximo":0},

```

```
10   "restricaoFuncoes":[{
11     "nome":"fatorial",
12     "retorno":true,
13     "recursiva":true,
14     "qtdParametros":1}]
15 }
```

Listagem B.10 – Especificação de três tipos de restrições.

Neste caso é exigido que a solução não utilize comandos if (podendo desenvolver a utilização do operador ternário), no máximo 1 comando *while* e uma função recursiva, com retorno e apenas um parâmetro e que se chama fatorial.

## B.2 Comunicação com a API

Este seção descreve como deve ser feita a comunicação com as principais classes da API.

### B.2.1 Classe AnalisadorEstatico

A classe *AnalisadorEstatico*, do pacote *analisador.estatico.ufs*, é a principal classe que deve ser utilizada para fazer a comunicação com a API. Ela possui um método público chamado *analisar()* que deve ser chamado logo após sua instanciação.

Para criar um objeto do tipo *AnalisadorEstatico* é necessário informar alguns parâmetros detalhados na quadro B.10:

| **Parâmetro** | **Valor** |
|---|---|
| linguagem | Um **inteiro** para indicar a linguagem de programação da solução. |
| código-fonte | Uma **string** com o código-fonte da solução do aluno. |
| json | Uma **string** com o json das restrições do problema. |

Quadro B.10 – Parâmetros do método construtor da classe AnalisadorEstatico

O método *analisar()* da classe *AnalisadorEstatico* retorna um objeto da classe *ResultadoAnalise* do pacote *analisador.estatico.ufs*. Este objeto possui um arquivo json com o resultado da análise das restrições, a quantidade de restrições passadas no arquivo *json* e a quantidade de restrições que não foram obedecidas pela solução do aluno.

```
1 package teste.analisador;
2
3 import java.io.File;
4 import java.io.IOException;
5 import java.nio.file.Files;
6 import analisador.estatico.base.Constantes;
```

```
7  import analisador.estatico.ufs.AnalisadorEstatico;
8  import analisador.estatico.ufs.ResultadoAnalise;
9
10 public class Main {
11
12     public static void main(String[] args) throws IOException {
13         File file_condigo_fonte = new File("solucao.py");
14         String codigo_fonte = new String(Files.readAllBytes(
       file_condigo_fonte.toPath()));
15
16         File file_restricao_json = new File("restricoes.json");
17         String restricao_json = new String(Files.readAllBytes(
       file_restricao_json.toPath()));
18
19         AnalisadorEstatico analisador = new AnalisadorEstatico(
       Constantes.PYTHON, codigo_fonte, restricao_json);
20         ResultadoAnalise resultado =  analisador.analisar();
21
22         String resultadoAnalise = resultado.getRetornoJson();
23         int totalRestricoes = resultado.getQtdRestricoes();
24         int restrInvalidas = resultado.getQtdRestricoesInvalidas();
25     }
26 }
```

Listagem B.11 – Exemplo de utilização da classe AnalisadorEstatico

A listagem B.11 apresenta um exemplo de como se acionar o analisador estático. Na linha 22 é retornada uma *String* com todos os detalhes da avaliação do analisador estático sobre o *código-fonte*. Os detalhes de como analisar essa mensagem de retorno estão nas próximas seções. Na linha 23 é retornada a quantidade total de restrições e na linha 24 é retornada a quantidade de restrições que não foram obedecidas. Por meio dessas duas últimas linhas, o cliente (juiz *on-line*) pode criar um mecanismo para penalizar o aluno de acordo com o percentual de restrições que não foram obedecidas.

### B.2.2 Classe DescricaoRestricao

A classe *DescricaoRestricao*, do pacote *analisador.estatico.ufs*, serve para traduzir um arquivo *json* em uma descrição que poderá ser utilizada no enunciado do problema.

Para instanciar um objeto do tipo *DescricaoRestricao* é necessário informar alguns parâmetros detalhados na quadro B.11:

Após criar o objeto do tipo *DescricaoRestricao*, basta chamar o método *getDescricao()* que será retornada uma lista de *strings* com uma descrição das restrições que foram passadas no

| Parâmetro | Valor |
|---|---|
| json | Uma **string** com o json das restrições do problema. |
| linguagem | Um **inteiro** para indicar a linguagem de programação da solução. |
| idioma | Um **inteiro** para representar em qual idioma as possíveis mensagens de *feedback* deverão estar. |

Quadro B.11 – Parâmetros do método construtor da classe DescricaoRestricao

arquivo *json*.

```
1  package teste.analisador;
2
3  package teste.analisador;
4
5  import java.io.File;
6  import java.io.IOException;
7  import java.nio.file.Files;
8  import java.util.List;
9
10 import analisador.estatico.base.Constantes;
11 import analisador.estatico.ufs.DescricaoRestricao;
12
13 public class Main2 {
14
15     public static void main(String[] args) throws IOException {
16         File file_restricao_json = new File("restricoes.json");
17         String restricao_json = new String(Files.readAllBytes(
    file_restricao_json.toPath()));
18         DescricaoRestricao descricao = new DescricaoRestricao(
    restricao_json, Constantes.PYTHON, Constantes.PORTUGUES);
19         List<String> descricoes = descricao.getDescricao();
20         for(int i = 0; i < descricoes.size(); i++){
21             System.out.println(descricoes.get(i));
22         }
23     }
24 }
```

Listagem B.12 – Exemplo de utilização da classe DescricaoRestricao

A listagem B.12 apresenta um exemplo de como utilizar a classe DescricaoRestricao. Supondo que o aquivo *json* passado foi o apresentado na listagem B.13, a lista de *strings* retornada pelo método *getDescricao()* será:

1. Você precisa utilizar exatamente 2 laços do tipo while;

2. Você precisa utilizar entre 1 e 3 desvios condicionais em sua solução;

3. Você precisa criar uma função chamada fatorial, com 1 parâmetro(s), recursiva e com retorno.

```
{
  "restricaoWhile":{
    "minimo":2,
    "maximo":2},

  "restricaoIf":{
    "minimo":1,
    "maximo":3},

  "restricaoFuncoes":[{
    "nome":"fatorial",
    "retorno":true,
    "recursiva":true,
    "qtdParametros":1}]
}
```

Listagem B.13 – Especificação de três tipos de restrições.

Para finalizar, é recomendável sempre utilizar a classe *Constantes*, do pacote *analisador.estatico.base*, para informar a linguagem de programação. Desta forma é possível evitar erros de consistência em futuras atualizações.

### B.2.3 Método getRetornoJson()

O método *getRetornoJson* (da classe *ResultadoAnalise* e do pacote *analisador.estatico.ufs*) retorna um texto (*string*) no formato *json* com o resultado da análise realizada. As próximas subseções descrevem como devem ser interpretados os resultados de todas as possíveis análises.

#### B.2.3.1 Retorno Funções

Suponha que tenha sido solicitada a análise das retrições da listagem B.14 no código-fonte da listagem B.15.

```
{"restricaoFuncoes":
    [{
  "nome":"subtrair",
  "retorno":true,
  "recursiva":false,
  "qtdParametros":1
  },

  {
```

```
10   "nome":"somar",
11   "retorno":true,
12   "recursiva":false,
13   "qtdParametros":1
14   },
15
16   {
17   "nome":"multiplicar",
18   "retorno":true,
19   "recursiva":false,
20   "qtdParametros":1
21   }]
22 }
```

Listagem B.14 – Especificação de restrições.

```
1 package huxleytest;
2 import java.util.Scanner;
3 public class HuxleyCode {
4     public static void subtrair(String args[]) {
5                         // alguma coisa
6     }
7 }
```

Listagem B.15 – Exemplo de solução

O resultado da análise será o json da listagem B.16.

```
1 {"retornoFuncoes":
2   [{
3   "existe":true,
4   "nome":true,
5   "parametros":true,
6   "recursao":false,
7   "retorno":false,
8   "foiChamada":false
9   },
10
11   {
12   "existe":true,
13   "nome":false,
14   "parametros":false,
15   "recursao":false,
16   "retorno":false,
17   "foiChamada":false
18   },
19
20   {
```

```
21    "existe":true,
22    "nome":false,
23    "parametros":false,
24    "recursao":false,
25    "retorno":false,
26    "foiChamada":false
27    }]
28 }
```

Listagem B.16 – Json de retorno para funções.

O primeiro detalhe que deve ser observado é que a ordem das restrições no json da listagem B.14 é mantida no json de retorno da listagem B.16. Dessa forma, o resultado da análise da função subtrair está apresentado entre as linhas 3 e 8, da função somar entre as linhas 12 e 17 e da função multiplicar entre as linhas 21 e 26 da listagem B.16.

O valor padrão de todos os elementos do json de retorno é *false* e eles devem ser analisados linha após linha.

A interpretação do resultado da função subtrair é: no código-fonte existe uma função com o nome desejado e que possui a quantidade desejada de parâmetros. Entretanto, não obedece às retrições de recursão e retorno, ou seja, neste caso, não possui retorno e não é recursiva. Vale ressaltar que o *false* nem sempre significa que a função não tem recursão ou retorno, mas sim que **não obedece às restrições de recursão e retorno**. Por exemplo, se fosse especificado que a função não deve ter retorno e a função tivesse o resultado também seria *false*.

A interpretação do resultado da função somar é: No código-fonte existe uma função, mas essa não se chama somar. Dessa forma, não se pode concluir mais alguma coisa a respeito da quantidade de parâmetros, retorno e recursão porque não existe uma função com o nome somar. O mesmo raciocínio é levado em consideração para a função multiplicar.

Para simplificar, só deve ser analisado o nome da função se ela existir. Só deve ser analisada a quantidade de parâmetros se existir uma função com o nome desejado. Só deve ser analisada a recursão e o retorno se a função tiver a quantidade de parâmetros desejada. O mesmo raciocínio serve para verificar se ela foi invocada.

#### B.2.3.2 Retorno Lista

O resultado da listagem B.17 indica que o código-fonte atendeu à restrição de lista.

```
1 {
2    "retornoLista":{
3    "atendeu":true
4 }}
```

Listagem B.17 – Json de retorno para lista.

#### B.2.3.3 Retorno Dicionário

Funciona da mesma maneira que o retorno de lista. O exemplo da listagem B.18 aponta que a restrição foi respeitada.

```
{
  "retornoDicionario":{
  "atendeu":true
}}
```

Listagem B.18 – Json de retorno para dicionário.

#### B.2.3.4 Retorno For

A listagem B.19 indica que o código-fonte não obedeceu à restrição *for* e que foram encontrados 4 *fors* na solução.

```

{"retornoFor":
  {"atendeu":false,
  "quantidade":4}
}
```

Listagem B.19 – Json de retorno para for.

#### B.2.3.5 Retorno While

Possui funcionamento semelhante à restrição *for*. No caso da listagem B.20 a restrição foi respeitada.

```

{"retornoWhile":
  {"atendeu":true,
  "quantidade":1}
}
```

Listagem B.20 – Json de retorno para while.

#### B.2.3.6 Retorno Desvios Condicionais

Possui funcionamento semelhante às restrições for e *while*. O exemplo da listagem B.21 aponta que a restrição realmente foi respeitada.

```

{"retornoIf":
  {"atendeu":true,
  "quantidade":1}
}
```

Listagem B.21 – Json de retorno para desvios condicionais.

#### B.2.3.7 Retorno Blocos Vazios

A listagem B.22 indica que o código não obedeceu à restrição de Blocos Vazios.

```
1
2 {"retornoBlocosVazios":
3     {"atendeu":false}
4 }
```

Listagem B.22 – Json de retorno para blocos vazios.

#### B.2.3.8 Retorno Chamada de Função

A listagem B.23 indica que a solução desejada está usando (se foi especificado para não usar) ou não está usando (se foi especificado para usar) uma determinada função.

```
1
2 {"retornoFuncaoChamada":
3 {
4     "atendeu":false
5 }}
```

Listagem B.23 – Json de retorno para chamada de função.

# C

# Evolução do Analisador Estático

Este apêndice apresenta os passos que devem ser seguidos para estender o Analisador Estático para se adaptar a novas linguagens de programação. As alterações se limitam a criar e alterar classes Java. Desta forma, não é necessário realizar alterações no formato das mensagens em JSON. Nas próximas seções serão apresentados os cinco passos para se estender o Analisador Estático.

## C.1 1º Passo - Alterar classe de Constantes

O primeiro passo para estender o Analisador Estático atual é criar uma nova constante com o nome da nova linguagem na classe *analisador.estatico.base.Constantes*. Atualmente há duas constantes: **PYTHON (valor 0)** e **JAVA (valor 1)**. Sempre que for necessário se referir a uma determinada linguagem é recomendado utilizar as constantes e não os seus respectivos valores.

## C.2 2º Passo - Criar nova ConcreteFactory

O segundo passo é criar uma nova ConcreteFactory específica para a nova linguagem. Esta nova classe deve ser criada no pacote *analisador.estatico.base* e deve implementar a interface LinguagemFactory. Por convenção, o nome das factories reflete o nome das linguagens, por exemplo: **PythonFactory** e **JavaFactory**. O diagrama da interface LinguagemFactory é apresentado na figura 47.

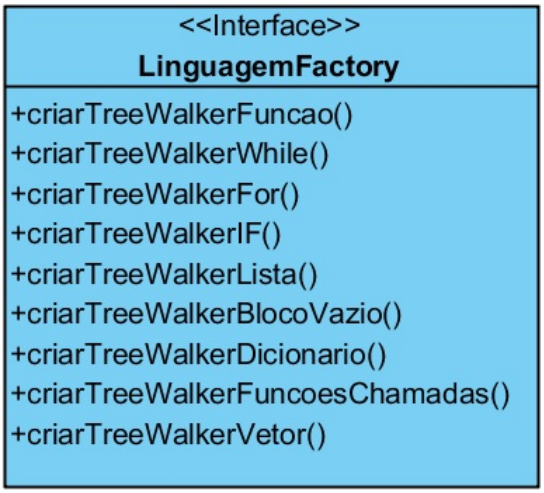


Figura 47 – Diagrama da interface LinguagemFactory

## C.3 3º Passo - Criar novo pacote

O terceiro passo é criar um novo pacote para que nele estejam concentrados todos os novos Tree Walkers da nova linguagem. Por convenção, o nome do pacote deve ser "analisador.estatico.tree.walker.**nomeDaLinguagem**".

## C.4 4º Passo - Criar novo Tree Walker

O quarto passo é referente à criação do Tree Walker. Todo Tree Walker (classe Java) deve estender a classe BaseVisitor da respectiva linguagem que foi gerada pelo ANTLR. Esta classe é fundamental para fazer o caminhamento pela Árvore Sintática Abstrata de alguma determinada submissão.

A classe **BaseVisitor** gerada pelo ANTLR para o Python 3 é **Python3BaseVisitor** e para o Java 7 é **JavaBaseVisitor**. Esses nomes são gerados de acordo com definições nas próprias gramáticas das referidas linguagens. Além disso, cada Tree Walker deve implementar uma interface que define um contrato do que deve ser de fato codificado.

Para fins de exemplo, o Tree Walker que se refere à função deve implementar no mínimo dois métodos: **getFuncoes()** e **visitarFuncao(ParserTree tree)**. O primeiro deve retornar todas as funções encontradas no código do aluno e o segundo é o ponto de partida para fazer o caminhamento sobre o código, por isso ele recebe uma árvore (**ParserTree**). Uma função é definida pela classe analisador.estatico.entidade.Funcao e representada pelo diagrama de classe da figura 48. A figura 49 possui todas as interfaces que devem ser implementadas pelos novos Tree Walkers da nova linguagem de programação de interesse.

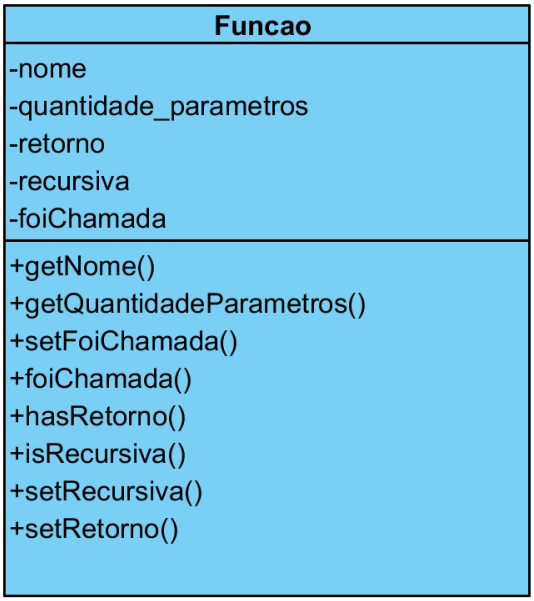


Figura 48 – Diagrama da classe Funcao

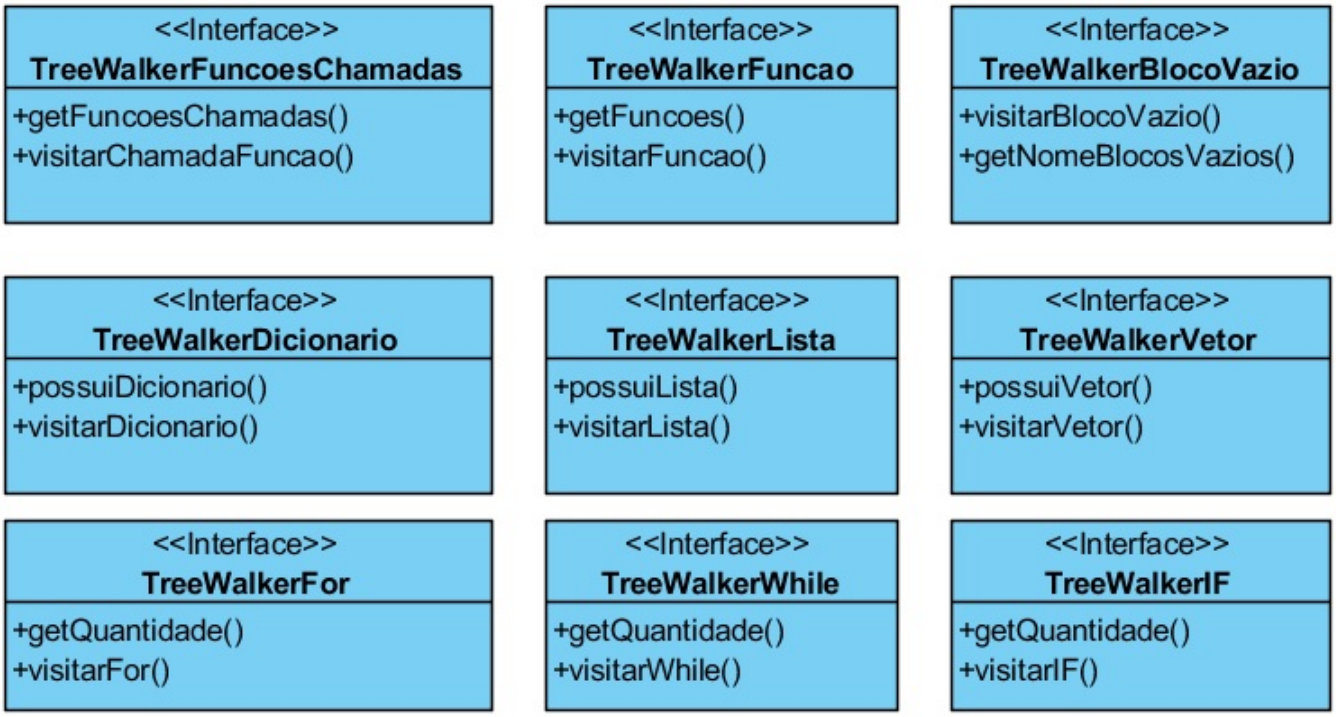


Figura 49 – Diagrama com todas as interfaces

## C.5 5º Passo - Alterar método criarFactoryEspecifica()

O quinto e último passo desta sequência é alterar o método **criarFactoryEspecifica()** da classe *analisador.estatico.ufs*. Este método é responsável por escolher a linguagem de programação em tempo de execução e retornar sua respectiva ConcreteFactory. Como apresentado no passo 2, todas as Factories específicas implementam **LinguagemFactory**. Desta forma, o método **criarFactoryEspecifica()** retorna uma **LinguagemFactory** que pode ser entendida como

qualquer uma das específicas durante todo o processamento.

A alteração deste método é simplemente adicionar uma condição para a nova linguagem de programação com a inicialização de componente léxico específico (ex: **JavaLexer**), da classe **CommonTokenStream** (lexer) e de seu determinado Parser (ex: **JavaParser**).

```
1
2  private LinguagemFactory criarFactoryEspecifica() {
3    ANTLRInputStream input = new ANTLRInputStream(codigo_fonte);
4    Lexer lexer = null;
5    if (linguagem == Constantes.PYTHON) {
6      lexer = new Python3Lexer(input);
7      CommonTokenStream tokens = new CommonTokenStream(lexer);
8      Python3Parser pythonParser = new Python3Parser(tokens);
9      parser = pythonParser.file_input();
10     return new PythonFactory();
11   } else if (linguagem == Constantes.JAVA){
12
13     lexer = new JavaLexer(input);
14     CommonTokenStream tokens = new CommonTokenStream(lexer);
15     JavaParser javaParser = new JavaParser(tokens);
16     parser = javaParser.compilationUnit();
17
18     return new JavaFactory();
19   }else{
20     // Nova linguagem
21     return null;
22   }
23 }
```

Listagem C.1 – Método criarFactoryEspecifica().